

Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorthesis

Friederike Löwe

Architektur-Reengineering einer Verifikationssoftware für Strukturmodelle

Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science
Department of Information and Electrical Engineering

Friederike Löwe

Architektur-Reengineering einer Verifikationssoftware für Strukturmodelle

Bachelorthesis eingereicht im Rahmen der Bachelorprüfung
im Studiengang Informations- und Elektrotechnik
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer : Prof. Dr. Thomas Lehmann
Zweitgutachter : Prof. Dr. Stefan Sarstedt

Abgegeben am 16. Februar 2012

Friederike Löwe

Thema der Bachelorthesis

Architektur-Reengineering einer Verifikationssoftware für Strukturmodelle

Stichworte

Software-Architektur, Neuentwicklung, Schichtenmodell, Komponenten, Schnittstellen, Software-Qualität, Testbarkeit, Erweiterbarkeit, Verständlichkeit

Kurzzusammenfassung

Die Aufgabe dieser Bachelorthesis ist es, die Architektur einer vorhandenen Software zur Verifikation von Strukturmodellen neu zu entwickeln. Um die Qualität der entwickelten Architektur zu belegen, wird ein Durchstich implementiert. Als wichtigste Qualitätskriterien gelten Testbarkeit, Erweiterbarkeit und Verständlichkeit der Architektur. Diese werden mit Hilfe einer klaren Strukturierung der Software in Schichten und Komponenten mit definierten Schnittstellen erreicht. Die vorliegende Arbeit dient der Dokumentation der Architektur und der Entscheidungen, die auf dem Weg zu dieser Lösung getroffen wurden.

Friederike Löwe

Title of the paper

Architecture reengineering of a verification-software for structural analysis models

Keywords

Software architecture, reengineering, layering, components, interfaces, software quality, testability, expandability, understandability

Abstract

This paper aims to describe the reengineering of the architecture of a verification-software for structural analysis models. To prove the quality of the developed architecture, a spike was implemented. The most important quality criteria are the testability, expandability and understandability of the software architecture. These are achieved by structuring the software with the aid of layers, components and clearly defined interfaces. The thesis at hand documents the new software-architecture and the decisions which have led to this solution.

INHALT

Abbildungen

TABELLEN

1. Einleitung

Diese Arbeit beschreibt die Neuentwicklung der Software-Architektur eines Programms zur Verifikation von Strukturmodellen im Offshorebereich. Als Basis für die Neuentwicklung dient eine existierende Software, die in ihrem Funktionsumfang erweitert und um zusätzliche Qualitätsmerkmale wie z. B. Erweiterbarkeit und Testbarkeit ergänzt werden soll. Nach einer Aufwandsabschätzung für eine Erweiterung der existierenden Software einerseits und einer Neuentwicklung auf Grundlage der bestehenden Funktionalität andererseits, wurde die Entscheidung zugunsten der Neuentwicklung getroffen.

Die neu entwickelte Architektur wird als Durchstich implementiert um damit exemplarisch zu belegen, dass alle geforderten Qualitätsmerkmale erfüllt werden. Ein Schwerpunkt der Anforderungen an die neue Software ist die einfache Erweiterbarkeit um neue Funktionen. Um die Erfüllung dieser Anforderung zu belegen, wird ein Funktionsbereich mit allen wesentlichen Strukturen implementiert, der als Vorlage für alle weiteren Funktionsbereiche dienen kann. Dadurch wird der Nachweis geführt, dass sich die Software leicht um diese Aspekte erweitern lässt, ohne dass grundlegende Änderungen in den Konzepten notwendig werden.

Eine Software-Architektur zu entwerfen, sollte ein zyklischer Prozess sein (Starke, 2011). Diese Arbeit beschreibt die entstandene Architektur, sowie die Entscheidungen und Rahmenbedingungen, die dazu geführt haben. In diese Entscheidungen sind alle derzeit geplanten und absehbaren Anforderungen an vollständige Software eingeflossen. Deshalb sollten keine grundlegenden Änderungen an der Architektur notwendig sein, um diese Anforderungen umzusetzen. Sollten weitere hinzukommen, die außerhalb des betrachteten Lösungsraumes liegen, sollte eine erneute Refaktorisierung der Architektur auf Basis der dokumentierten Architekturentscheidungen problemlos möglich sein.

1.1 Das Unternehmen

Diese Arbeit entsteht bei der Overdick GmbH & Co. KG, einem Ingenieursbüro, in dem Bauingenieure und Schiffbauingenieure unter anderem Simulationsmodelle von Offshoreplattformen entwerfen. Aktuell bearbeiten hier 53 Mitarbeiter Entwicklungsaufträge, bei denen die Kunden von der Entwicklung bis zur Installation einer Offshoreplattform begleitet werden.

1.2 Der Geschäftsprozess

Wenn ein Kunde, beispielsweise eine Plattform in Auftrag gibt, so entwickeln die Ingenieure zunächst anhand der identifizierten Anforderungen ein simulationsfähiges Strukturmodell. Dieses entsteht innerhalb einer Software, die speziell für diesen Einsatzzweck konzipiert ist; zum Beispiel Staus7 (Strand7 Pty Ltd). Mit dieser Software ist es möglich, die Stabilität des Modells zu prüfen, indem man Lasten simuliert, die darauf einwirken. Als Simulationsergebnisse erhalten die Ingenieure Daten über Verformungen und Kräfte, anhand derer sie einschätzen können, wie sich die Plattform später unter dem Einfluss von Winden und Wellen verhalten wird. Welche Kräfte auftreten können und wie die Ergebnisse zu beurteilen sind, ist in verschiedenen Normen festgelegt, die eingehalten werden müssen. Ergibt die Simulation, dass die Plattform den aufgebrachten Lasten nicht standhalten können wird, müssen die Ingenieure ihren Entwurf überarbeiten und die Simulation später erneut durchführen.

1.3 Die Problemstellung

Bisher werden die Lasten, mit denen das Strukturmodell geprüft werden soll, per Hand in das Simulationsprogramm eingegeben. Da dieser Vorgang zeitaufwendig ist, werden die Modelle nur so genau wie nötig geprüft. Bei der manuellen Lastaufbringung nehmen die Ingenieure beispielsweise für die Wellenlasten räumlich konstante Maximalwerte an, was bedeutet, dass eine sehr pessimistische Einschätzung der Haltbarkeit der Plattform entsteht. Dadurch müssen die Modelle oft unnötig stark ausgelegt werden, auch über einen eingeplanten Sicherheitsfaktor hinaus. Dies führt zu erhöhten Kosten für Arbeitszeit und Material. Teilweise ist mit dieser Herangehensweise gar kein Stabilitätsnachweis möglich.

Besser wäre es, Wellenberge und -täler als entsprechende Lasten so über das Modell zu verteilen, wie sie tatsächlich auftreten können. Damit wird die Berechnung genauer und die Konstruktionen werden wirtschaftlicher. Auf diese Weise würde die Bestimmung der Lasten aber so komplex, dass die Ingenieure deutlich länger als bisher brauchen würden, um die Daten in das Simulationsprogramm einzugeben. Die Kosten, die durch die manuelle Eingabe der genauen Lasten entstehen würden, könnten aber durch die Materialeinsparung nicht gerechtfertigt werden.

Aus diesem Grund soll eine Anwendung entwickelt werden, die automatisch die Verteilung der Lasten für eine Welle über die räumliche und zeitliche Dimension berechnet und diese Lastfälle automatisiert an das Simulationsprogramm über-

trägt, sodass die Rechnung insgesamt genauer und die Bearbeitung schneller wird. So könnte Arbeitszeit eingespart werden, weil die Lasten jetzt automatisch generiert würden. Gleichzeitig könnte die Rechnung beliebig genau programmiert werden, wodurch Material in der Konstruktion eingespart werden könnte, weil die Berechnungen nicht mehr so pessimistisch abgeschätzt werden.

1.4 Bestehender Lösungsansatz

Es wurde bereits ein Programm entwickelt, das diese und ähnliche Aufgaben zur Optimierung des Entwicklungsablaufes übernehmen soll. Das Programm schließt an die vorhandene Simulationssoftware Straus7 an und wird als „Straus API-Tool“ bezeichnet. API steht für „Application Programming Interface“ und bezieht sich auf die Programmierschnittstelle von Straus7.

Damit sieht der Geschäftsprozess so aus, dass das konstruierte Straus-Modell in das API-Tool geladen wird und aus diesem Programm, anhand von eingestellten Parametern, Lasten generiert werden, die anschließend automatisch in der Modell-Datei gespeichert werden. Wird die Straus-Modell-Datei in Straus7 geöffnet, kann die Simulation mit den generierten Lasten gestartet werden. Der Aufwand für das Aufbringen der Lasten hat sich erheblich verringert.

Bei der Benutzung des Straus API-Tools treten allerdings unspezifische Fehler auf, die häufige Neustarts des Programms erforderlich machen. Eine Verbesserung der Benutzbarkeit durch einen systematischen Testansatz erweist sich als schwierig, da die Programmstruktur keine konsequente Kapselung und Modularisierung enthält. Das Beheben von Fehlern ist aufwendig, da der Quellcode lückenhaft dokumentiert ist.

Nach softwaretechnischer Analyse des vorhandenen Programms wurde davon Abstand genommen, das Programm zu erweitern oder zu verbessern. Stattdessen wurde gemeinsam mit dem bisherigen Entwickler der Entschluss getroffen, das Programm von Grund auf neu zu konzipieren und zu implementieren.

1.5 Aufgabenstellung

Die Architektur des neuen Programms ist Gegenstand dieser Arbeit. Sie soll einerseits alle geforderten Funktionen ermöglichen, andererseits aber auch einer Reihe nicht-funktionaler Anforderungen gerecht werden, die später in diesem Kapitel benannt werden. Als Qualitätstest für die Architektur wird ein Durchstich imple-

mentiert, der eine möglichst repräsentative Auswahl der geforderten Funktionen enthält. Das neue Programm wird als „Straus Multitool" bezeichnet. Es soll langfristig das Straus API Tool ersetzen. Dadurch wird der Geschäftsprozess nicht weiter verändert, das Prinzip des Programms bleibt bestehen.

Für das Straus Multitool wurde eine Anforderungsanalyse durchgeführt. In Anhang A.1 sind die Ergebnisse als Menge von Anforderungen zusammengefasst. Die Dokumentation einer Anforderung liegt in folgendem Format vor:



REQ-023	Name der Anforderung	Durchstich / später
		Quelle

Beschreibung der Anforderung in einem Fließtext…

Die Anforderungen haben eine eindeutige Nummer in der Form *REQ-xxx* und einen eindeutigen Namen. Im Feld rechts oben steht die Einordnung, zu welchem Zeitpunkt die Anforderung berücksichtigt wird, also ob sie im Rahmen dieser Arbeit umgesetzt wird (Durchstich) oder später. Das Feld „Quelle" beinhaltet die Information, von wem oder von welcher Stelle die Anforderung stammt. Das kann ein Meetingprotokoll, das Straus API Tool oder das Kürzel eines Overdick-Mitarbeiters sein.

1.5.1 Funktionalität

In diesem Abschnitt wird der im Rahmen dieser Arbeit zu implementierende Funktionsumfang beschrieben. Aspekte, die für Architektur-Entscheidungen wichtig sind, aber nicht für die Durchstich-Implementierung vorgesehen sind, werden hier nicht beschrieben. Sie finden sich in den Anforderungen und werden in den späteren Kapiteln bei der Begründung der entsprechenden Architektur-Entscheidung referenziert.

1.5.1.1 Persistenz

Das Programm soll die Möglichkeit bieten, Informationen wie die Konfiguration des Programms oder Parameter für Berechnungen zu speichern. Die Werte sollen nach einem Neustart des Programms wieder hergestellt werden. Diese Anforderung ist nach der Analyse hinzugekommen, betrifft aber auch *REQ-007*.

1.5.1.2 Modellverwaltung

Das Programm soll mit Straus-Modelldateien arbeiten können. Es soll in der Lage sein, sie zu öffnen, zu lesen, zu bearbeiten, zu speichern und zu schließen (siehe

REQ-019, *REQ-026* und *REQ-50*). Dazu gehört insbesondere, dass neue Lasten in der Datei gespeichert werden (*REQ-020* und *REQ-022c*), damit ihre Auswirkungen später mit Hilfe von Straus7 simuliert werden können.

1.5.1.3 Windlasten

Um eine der geplanten Berechnungen umzusetzen, soll das Programm Windlasten generieren können (*REQ-022* und *REQ-022a*), die es aus vom Benutzer eingegebenen Parametern wie Windgeschwindigkeit und Windrichtung berechnet. Dies kann für Modelle durchgeführt werden, die mit Stäben und Verbindungsknoten modelliert wurden. Pro Stab wird dabei eine Streckenlast berechnet, also eine als linear angenommene Größe, die Auskunft auf die Last auf einem bestimmten Längenstück des Stabes gibt.

Die Lasten sollen als Werte in einer Tabelle und als mathematischer Graph dargestellt werden (*REQ-022b*). Die Tabelle soll dazu Eingangsparameter und Ergebnisse der Rechnung für jeden Stab im Modell anzeigen. Der Graph zeigt für Stäbe mit gleichem Querschnittsprofil eine Verteilung der Lasten über die Höhe an.

1.5.1.4 Protokollierung

Das neue Programm soll während des Durchlaufs wichtige Schritte in einer Logging-Datei protokollieren (*REQ-006*). Damit sollen auch eventuell auftretende Programmfehler leicht lokalisiert werden können, die erst bei der Berechnung durch die Ingenieure auffallen.

1.5.1.5 Undo und Redo

Es soll prinzipiell möglich sein, Befehle per Undo- und Redo-Schaltfläche zurückzunehmen und zu wiederholen (*REQ-057*). Für welche Befehle dies erforderlich ist, ist im Einzelfall zu entscheiden.

1.5.2 Nicht-funktionale Aspekte

Bei der Entwicklung des neuen Programms sind die Aspekte Erweiterbarkeit, Wartbarkeit und Testbarkeit besonders wichtig. Eine Unterteilung in Komponenten macht das Programm leichter erweiterbar. Besonders für das Hinzufügen von neuen Berechnungsmöglichkeiten sollen leicht Module hinzugefügt werden können. Unter dem Begriff „Modul“ soll im Rahmen dieser Arbeit ein Paket von Funktionen verstanden werden, die eine gemeinsame Aufgabe erfüllen und das vorhandene Programm erweitern (*REQ-001*).

Bei der Testbarkeit wird vor Allem auf die leichte Anwendbarkeit von Unit-Tests geachtet. Die Struktur des Quellcodes muss also möglichst geringe Abhängigkeiten aufweisen, damit einzelne Elemente leicht isoliert werden können und einzeln in ein Test-Korsett eingespannt werden können.

Für die Wartbarkeit der Software ist es wichtig, dass sowohl die Struktur des Programms als auch der Quellcode leicht verständlich ist, damit die Wahrscheinlichkeit für Fehler so weit reduziert werden kann wie möglich. Die Code-Dokumentation (*REQ-008*) soll möglichst hilfreich und umfassend sein, darf aber nicht zu lang werden, damit sie leicht wartbar bleibt – vgl. dazu (McConnell, 2004).

1.5.3 Merkmale des Straus Multitools

Um eine passende Architektur für das Straus Multitool entwerfen zu können, ist es wichtig, sich über die geplanten Merkmale des Programms klar zu werden. Dabei konnten folgende Merkmale gefunden werden:

- *Offenes System:* Das Programm soll durch Module erweitert werden können.
- *Applikation:* Das Programm wird direkt von Menschen genutzt.
- *(Nicht) terminierend:* Das Programm soll als Sammlung von Werkzeugen so lange zur Verfügung stehen, bis der Benutzer das Programm wieder schließt. Das Programm ist also nicht terminierend. Module können allerdings auch als Transformation konzipiert werden, damit wären diese dann terminierend.
- *Reaktives System:* Das Programm hält andauernde Kommunikation mit dem Benutzer aufrecht.

Außerdem kann man sagen, dass es sich um ein *Einbenutzersystem* handelt. Die Module sind teilweise sehr *berechnungsintensiv*. Das Programm muss aber *keine Echtzeitanforderungen* erfüllen.

1.6 Aufbau der Arbeit

Diese Arbeit ist wie folgt aufgebaut: Im nächsten Kapitel werden Qualitätskriterien für Software-Architekturen eingeführt und es wird beschrieben, wie diese Kriterien für das Straus Multitool erreicht werden sollen. Außerdem wird eine Übersicht über die neue Software-Architektur gegeben und die Entscheidungen, die zu dieser Lösung geführt haben, werden dokumentiert und begründet.

Das dritte Kapitel beschreibt einzeln die Bausteine, aus denen die Architektur zusammengesetzt ist, und ihre Funktion aus der Innenansicht. Auch auf dieser Ebene werden getroffene Entscheidungen zum Nachvollziehen festgehalten. Die Aspekte Implementierung und Test werden für jede Komponente einzeln beschrieben, sofern dies für die Architektur wichtig ist.

Im vierten Kapitel werden technische Konzepte beschrieben, die Komponenten übergreifend für die Architektur relevant sind. Auch Aspekte, die auf Design- und Implementierungsebene für alle Komponenten wichtig sind, werden hier vorgestellt.

Mit dieser Abschlussarbeit hat die Implementierung der neuen Software erst begonnen. Deshalb werden in Kapitel fünf die bisher erreichten Arbeitsschritte zusammengefasst und bewertet. Außerdem wird ein Ausblick auf die zukünftige Entwicklung gegeben.

2 Die Architektur

Nach Len Bass, Paul Clements und Rick Kazman ist die Software-Architektur eines Programms dessen Struktur. Dazu gehören die Software-Elemente, deren von außen sichtbare Schnittstellen und die Beziehungen dieser Elemente untereinander (Bass, et al., 2003). Weiterhin sagen die Autoren, dass Architektur die höchste Ebene des Designs ist. Damit umfasst der Begriff „Design“ den vollständigen Entwurf, die Architektur ist ein Teil des Designs. Nach dieser Definition soll in diesem Kapitel die Architektur der neuen Anwendung beschrieben werden. Alle weiteren Designentscheidungen finden Platz in den Kapiteln 3 und 4.

Zum Entwurf der Architektur ist es wichtig, sich über die Systemgrenzen klar zu sein. Dazu zeigt Abbildung 1 die Kontextsicht des Programms. Es sind das Nachbarsystem Straus7, die Zusammenhänge zu Dateien, sowie die Interaktionen mit den Benutzern dargestellt. Der Zugriff auf die Straus-Dateien kann über die Straus-API erfolgen, deshalb ist kein direkter Zugriff vorgesehen sondern es wird auf die vom Hersteller zur Verfügung gestellten Funktionen zurückgegriffen.

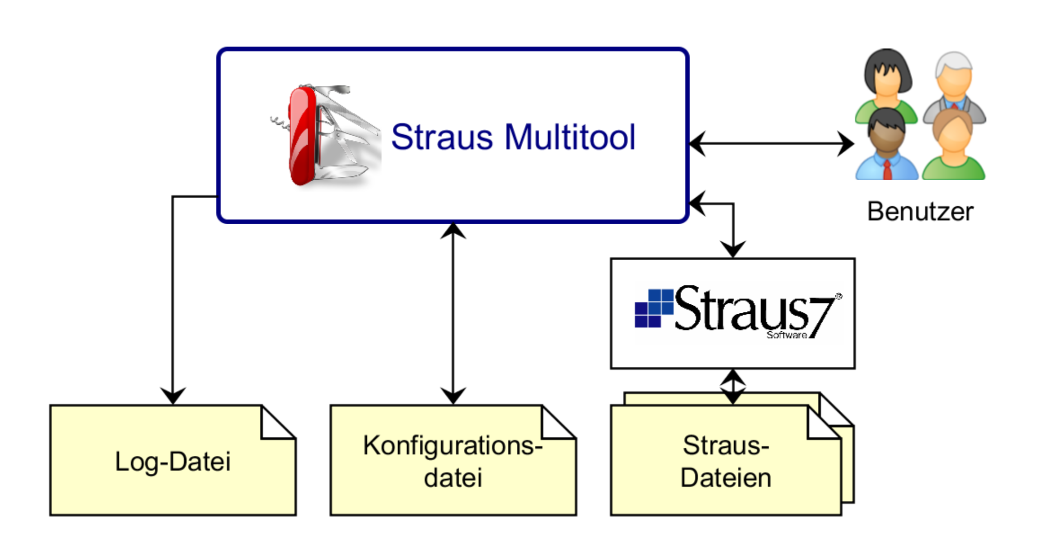


Abbildung 1: Kontextsicht des neuen Programms

In diesem Kapitel wird zunächst beschrieben, wie die Qualität von Software-Architektur definiert werden kann und wie sie für das Straus Multitool sichergestellt werden soll. Anschließend wird die entwickelte Architektur vorgestellt und erläutert.

2.1 Qualität der Software-Architektur

Um die Qualität von Software zu bewerten gibt die DIN/ISO 9126 die Kriterien Funktionalität, Benutzbarkeit, Effizienz, Änderbarkeit, Übertragbarkeit und Austauschbarkeit an. Diese sind je nach Aufgabe der Software unterschiedlich schwierig umzusetzen und unterschiedlich relevant. Gernot Starke fügt in seinem Buch „Effektive Software-Architekturen“ (2011) den in der Norm genannten Kriterien noch Verständlichkeit und Testbarkeit hinzu. Tabelle 1 zeigt die Definitionen der Begriffe aus der Norm.

Funktionalität	Vorhandensein von Funktionen mit festgelegten Eigenschaften; diese Funktionen erfüllen die definierten Anforderungen
Zuverlässigkeit	Fähigkeit der Software, ihr Leistungsniveau unter festgelegten Bedingungen über einen festgelegten Zeitraum zu bewahren
Benutzbarkeit	Aufwand, der zur Benutzung erforderlich ist, und individuelle Beurteilung der Benutzung durch eine festgelegte oder vorausgesetzte Benutzergruppe
Effizienz (Performance)	Verhältnis zwischen dem Leistungsniveau der Software und dem Umfang der eingesetzten Betriebsmittel unter festgelegten Bedingungen
Änderbarkeit	Aufwand, der zur Durchführung vorgegebener Änderungen notwendig ist. Änderungen: Korrekturen, Verbesserungen oder Anpassungen an Änderungen der Umgebung, der Anforderungen und der funktionalen Spezifikationen
Übertragbarkeit	Eignung der Software, von einer Umgebung in die andere übertragen zu werden.
Austauschbarkeit	Möglichkeit, diese Software anstelle einer spezifizierten anderen in der Umgebung jeder Software zu verwenden, sowie der dafür nötige Aufwand

Tabelle 1: Qualitätskriterien der DIN/ISO 9126 (Starke, 2011)

In den folgenden Abschnitten wird auf die einzelnen Kriterien näher eingegangen: Alle genannten Kriterien werden genauer beschrieben, ihre Relevanz für die zu entwickelnde Anwendung wird beurteilt und es wird erläutert, welche Maßnahmen ergriffen werden, um das Qualitätskriterium zu erfüllen. Die Relevanz der Kriterien wird über eine Bewertungsskala von *A* = *„wichtig“* über *B* = *„mittel“* bis zu *C* = *„weniger wichtig“* eingestuft.

2.1.1 Verständlichkeit

Verständlichkeit ist eine wichtige Eigenschaft (Relevanz: A) einer guten Software-Architektur. Dazu schreibt Kent Beck in seinem Buch „Implementation Patterns" (2008): „Kommunikation und Einfachheit arbeiten oftmals Hand in Hand. Je geringer die Komplexität ist, desto leichter lässt sich ein System verstehen. Je mehr Sie sich auf Kommunikation konzentrieren, desto einfacher ist es zu sehen, welche Komplexität verworfen werden kann."

Für das Straus API-Tools existiert keine Strukturdokumentation, die das Verständnis erleichtern könnte. Deshalb wurde über die Funktion *Code Visualization* in Visual Studio eine Übersicht über die Abhängigkeiten zwischen den einzelnen Klassen automatisch generiert. Das Ergebnis findet sich in Abbildung 2. Dabei ist die Dicke einer Verbindungslinie proportional zur Abhängigkeit der beiden Klassen, die die Linie verbindet. Abhängigkeiten können zum Beispiel Aufrufe oder Referenzen sein.

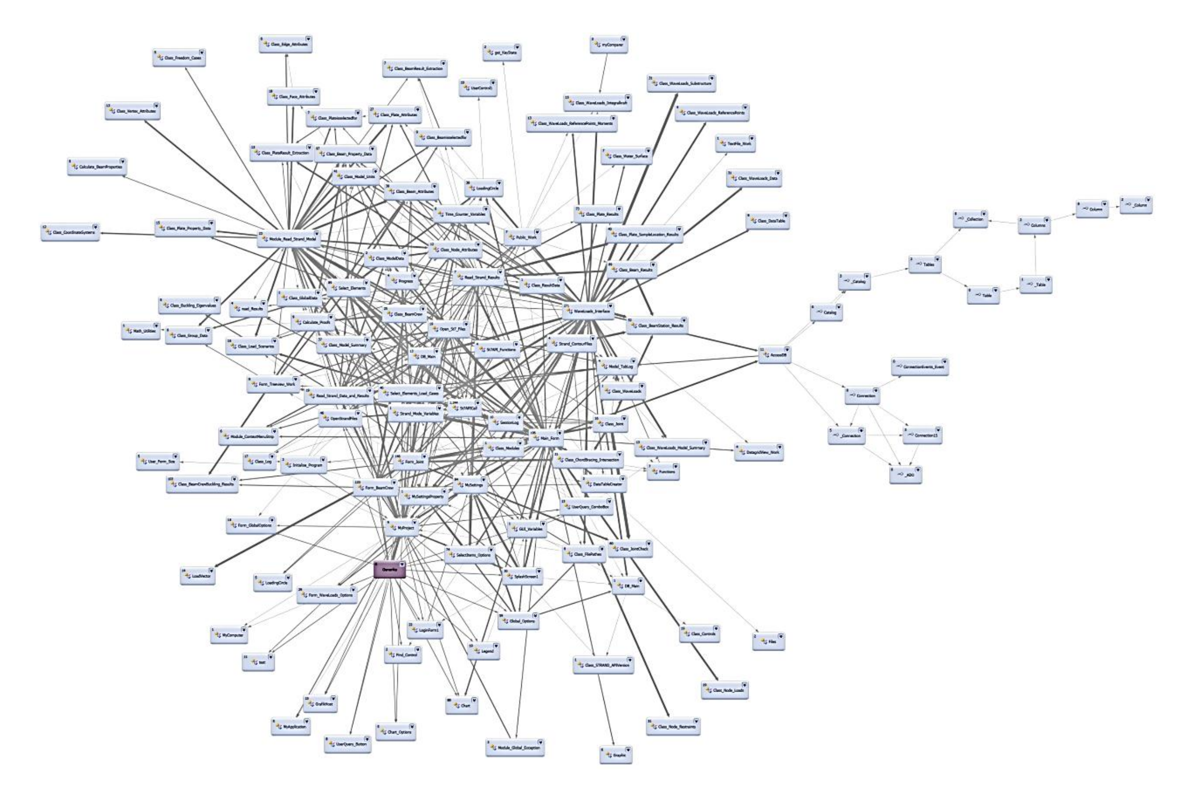


Abbildung 2: Abhängigkeiten der Klassen innerhalb des Straus API-Tools

Der Nutzen dieser Visualisierung ist begrenzt, da keine Struktur der Klassen untereinander erkennbar ist. Leider bietet die Software nicht die Möglichkeit, die Anordnung der Klassen zu verändern, um eine Gruppierung zu erreichen. Dieses Bild macht aber deutlich, dass eine verständliche Architektur-Dokumentation Klassen in Funktionsgruppen aufteilen sollte. Auch ist es für die Übersichtlichkeit wichtig, die Abhängigkeiten möglichst gering zu halten und die Verantwortlich-

keiten für verschiedene Aufgaben auf die Klassen nach einer konsistenten Logik aufzuteilen; vgl. (Wirfs-Brock, et al., 2002).

Um eine Ordnung in die Klassen des Straus Multitools zu bringen, werden diese in Komponenten gruppiert. Diese wiederum werden auf verschiedene Schichten aufgeteilt (siehe Kapitel 2.3 „Die Komponenten").

2.1.2 Testbarkeit

Bei jeder Software ist Testbarkeit ein wichtiges Kriterium. Gernot Starke empfiehlt, dass Testbarkeit von Anfang an zu den wichtigsten Kriterien für den Systementwurf gehören sollte (Starke, 2011 S. 174). Dies muss nicht zwangsläufig bedeuten, dass nach dem Test-Driven-Development-Prinzip entwickelt wird, sondern vor allem, dass ein testbares Design garantiert ist. Dies definiert Roy Osherove als ein Design, in dem für jedes logische Code-Stück leicht und schnell ein Unit-Test geschrieben werden kann. (Osherove, 2010)

Für das Straus Multitool ist Testbarkeit neben Wartbarkeit eines der beiden Argumente für die Neuentwicklung. Denn die Software entsteht in einer Domäne, in der die Korrektheit der Berechnungen relevant für die Sicherheit ist (*REQ-039*). Würde die Software Rechenfehler bei den Lastenrechnungen machen, die Ingenieure würden sich aber auf die Software verlassen und das damit berechnete Modell so bauen lassen, könnte das spätere Bauwerk einstürzen und dabei Leben gefährden. Damit lässt sich nach der Definition von Andreas Spillner und Tilo Linz der für dieses Projekt wichtigste Teil des Testens mit Unit-Tests abdecken, da sie deren wichtigste Aufgabe in der Sicherstellung sehen, „dass das jeweilige Testobjekt die laut seiner Spezifikation geforderte Funktionalität korrekt und vollständig realisiert." (Spillner, et al., 2003)

Darüber hinaus sind aber auch sogenannte Akzeptanztests eine gute Möglichkeit, die Qualität der Software sicher zu stellen. Sie prüfen, ob das getestete Stück Code die richtige Aufgabe erledigt. Diese Art von Test wird sicherlich später hilfreich, wenn die Software umfangreicher geworden ist. Momentan wird die Akzeptanz pragmatisch angegangen, indem der geschriebene Code per Review mit dem Entwickler des Straus API-Tools durchgegangen wird. Dieser ist als Reviewer deshalb gut geeignet, da er als Entwickler den Code versteht und als Bauingenieur ebenfalls Nutzer des Straus Multitools wird und deshalb die Anforderungen und deren Umsetzung sehr genau einschätzen kann.

Für die Architektur-Entscheidungen ist hier also nur der Unit-Test relevant. Mit Hilfe von verschiedenen Techniken und Mustern wurde die Isolierbarkeit von Klassen und Methoden sichergestellt, sodass diese von Testmethoden nutzbar wurden. Die Techniken und Muster werden jeweils in dem Kapitel über den Architekturbaustein beschrieben, in dem sie verwendet werden.

2.1.3 Funktionalität

Die Gewährleistung der richtigen Funktionalität eines Programmes ist sehr wichtig (Relevanz: A). Nach der DIN/ISO 9126 kann das Qualitätskriterium Funktionalität zum Beispiel die Aspekte Angemessenheit, Richtigkeit, Interoperabilität und Ordnungsmäßigkeit umfassen. Die Beschreibung dieser Aspekte ist in Tabelle 2 zusammengefasst.

Angemessenheit	Liefern der richtigen oder vereinbarten Ergebnisse oder Wirkungen, z. B. die benötigte Genauigkeit von berechneten Werten
Richtigkeit	Eignung der Funktionen für spezifizierte Aufgaben, z. B. aufgabenorientierte Zusammensetzung von Funktionen aus Teilfunktionen
Interoperabilität	Fähigkeit, mit vorgegebenen Systemen zusammenzuwirken. Hierunter fällt auch die Einbettung in die Betriebsinfrastruktur.
Ordnungsmäßigkeit	Erfüllung von anwendungsspezifischen Normen, Vereinbarungen, gesetzlichen Bestimmungen und ähnlichen Vorschriften

Tabelle 2: Aspekte des Qualitätsmerkmals „Funktionalität“ (Starke, 2011)

Die **Angemessenheit** der Funktionalität wird ebenso wie die **Richtigkeit** durch regelmäßige persönliche Gespräche innerhalb der Planungsphase und ausführlichen Reviews innerhalb der Durchführungsphase sichergestellt. Für die Angemessenheit existieren keine quantitativen Anforderungen seitens der Firma. Es werden aber Ideen festgehalten, wie die Genauigkeit der Rechnungen langfristig verbessert werden kann. Da die Rechnungsgenauigkeit nicht im Fokus dieser Arbeit liegt, werden diese noch nicht implementiert.

Für den Aspekt **Interoperabilität** ist wichtig, dass das Programm sowohl mit Straus7 zusammenarbeiten soll, als sich auch langfristig in den betrieblichen Ablauf bei Overdick einpassen soll. Das Programm soll eng mit der Software Straus7 zusammenarbeiten. Dazu stellt die Firma Strand7 eine Programmierschnittstelle bereit, die mit Delphi, C/C++, Matlab, VisualBasic (verschiedene Versionen und

VBA) und Fortran angesprochen werden kann, vgl. (Straus7 Software, 2010). Der Entwickler des Straus API Tools hat sich für die Programmiersprache Visual Basic .NET entschieden. Um sich auch auf die Kenntnisse dieses Entwicklers stützen zu können und diesen später bei der Implementierung mit einbeziehen zu können, soll weiterhin mit dem .NET-Framework entwickelt werden. Dieses unterstützt mehrere Sprachen, die über eine gemeinsame Zwischenschicht problemlos miteinander kommunizieren können; vgl. (MSDN Library, 2012).

Um die Sprache festzulegen, mit der das neue Programm entwickelt werden soll, spielt hier die Verbreitung bzw. Erlernbarkeit der Sprache eine wichtige Rolle, damit mögliche Neuzugänge des Entwicklerteams möglichst schnell Verständnis für den vorhandenen Code erlangen können. Das Unternehmen TIOBE Software stellt online einen Index über die Verbreitung der einzelnen Programmiersprachen zur Verfügung. Der Index wird monatlich neu berechnet – anhand der Trefferzahl, die populäre Suchmaschinen zu diesem Thema erzeugen. Hier steht C# in 2011/12 ständig unter den ersten fünf Sprachen (zusammen mit Java, C, C++, Objective-C und PHP), vgl. Abbildung 3: Tiobe Programming Community Index.

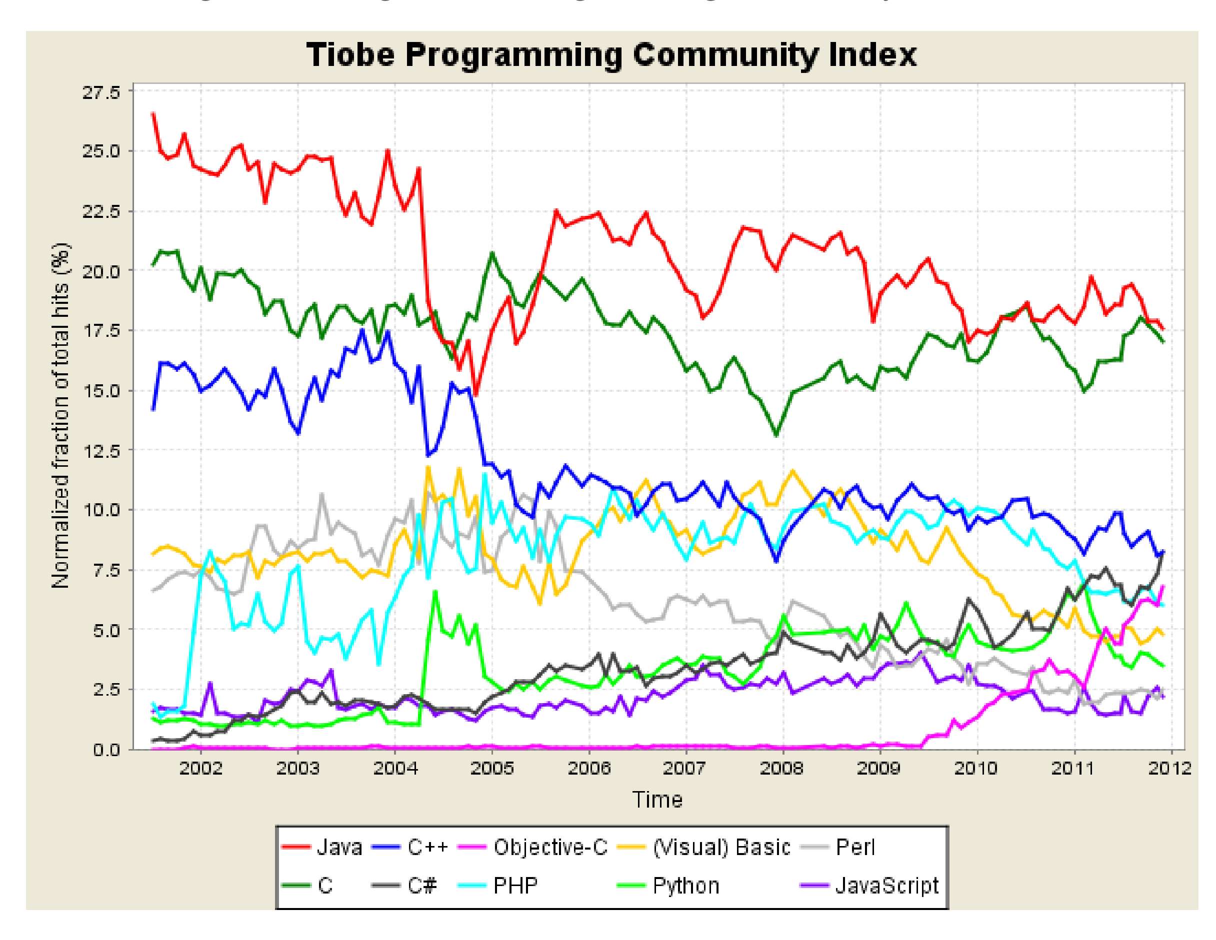


Abbildung 3: Tiobe Programming Community Index (TIOBE Software)

Visual Basic .NET hingegen findet sich z. B. im Februar 2012 auf Platz 16, aufgerückt von Platz 24, der Rangordnung (TIOBE Software). Hinzu kommt, dass C# mit Kenntnissen einer der anderen Sprachen der Top5 sehr leicht zu lernen ist, weil die Syntax sich nicht stark unterscheidet.

Die **Ordnungsmäßigkeit**, also die Erfüllung von anwendungsspezifischen Normen stellt keine architektonische Herausforderung dar. Normen der Softwareentwicklung müssen nach Vorgabe von Overdick nicht eingehalten werden. Normen für die Lastenrechnungen fließen in die Algorithmen zur Berechnung ein und müssen in der Umsetzung sehr penibel getestet werden, spielen aber darüber hinaus aus Architektursicht keine Rolle.

2.1.4 Zuverlässigkeit

Als Aspekte des Themas „Zuverlässigkeit" werden in der DIN/ISO 9126 die Begriffe Reife, Fehlertoleranz und Wiederherstellbarkeit vorgeschlagen. In Tabelle 3 werden die Begriffe näher erläutert. Von Overdick wurde der Zuverlässigkeit des Programms nur eine geringe Priorität beigemessen (Relevanz: C).

Reife	Geringere Versagenshäufigkeit durch Fehlzustände
Fehlertoleranz	Fähigkeit, ein spezifiziertes Leistungsniveau bei Software-Fehlern oder Nicht-Einhaltung ihrer spezifizierten Schnittstelle zu bewahren
Wiederherstellbarkeit	Fähigkeit, bei einem Versagen das Leistungsniveau wiederherzustellen und die direkt betroffenen Daten wiederzugewinnen.

Tabelle 3: Aspekte des Qualitätsmerkmals „Zuverlässigkeit" (Starke, 2011)

Eine Form von **Reife** kann innerhalb einer Durchstich-Implementierung nicht erreicht werden. Sie ist ein langfristiges Ziel, das durch gute Testbarkeit, gründliche Reviews und übersichtliches Design angestrebt wird.

Fehlertoleranz ist in diesem System nicht vorgesehen. Beinhaltet das Modell Fehler oder ist ein Teilsystem nicht zugreifbar, soll dieser Zustand zunächst behoben werden, bevor das Programm mit der eigentlichen Rechnung beginnt. Dazu wird der Nutzer möglichst präzise über den aufgetretenen Fehler unterrichtet.

Der Fähigkeit zur **Wiederherstellung** von Zuständen des Modells wurde eine eher geringere Priorität beigemessen, da die Modelldatei so lange unverändert bleibt, wie der Benutzer nicht auf „speichern" drückt.

2.1.5 Benutzbarkeit

Der Benutzbarkeit wird von Overdick eine untergeordnete Relevanz zugeordnet (Relevanz: B), weil die Meinung besteht, dass der Einarbeitungsaufwand nicht so wichtig ist, wie richtige Funktionalität. Unter dem Thema Benutzbarkeit finden sich in der DIN/ISO 9126 folgende Aspekte:

Verständlichkeit	Aufwand für den Benutzer, das Konzept der Anwendung zu verstehen
Erlernbarkeit	Aufwand für den Benutzer, die Anwendung zu erlernen (z. B. Bedienung, Ein-, Ausgabe)
Bedienbarkeit	Aufwand für den Benutzer, die Anwendung zu bedienen

Tabelle 4: Aspekte des Qualitätsmerkmals „Benutzbarkeit" (Starke, 2011)

Die drei Kriterien für gute Benutzbarkeit (siehe auch *REQ-012*) sollen mit verschiedenen Aktivitäten erreicht werden:

- Es wird ein Benutzerhandbuch erstellt (*REQ-003*), das dem Ingenieur hilft, die Funktionen des Programms leicht zu finden und genau zu verstehen, was eine bestimmte Funktion tut (*REQ-009*), damit er einschätzen kann, wann er die Funktion einsetzen kann und mit welchen Ergebnissen und Toleranzen er zu rechnen hat.
- Es wird gemeinsam mit den Nutzern versucht, die Funktionsaufrufe in Buttons oder Menüs so zu platzieren, dass Funktionen im gleichen Kontext in der Programmoberfläche nah beieinander liegen.
- Zur Oberflächenstruktur des Programms wurde ein Usability-Experte zu Hilfe gerufen, der die aktuelle Aufteilung als positiv und hilfreich eingeschätzt hat. Ob die Überlegungen richtig waren, kann sich nur langfristig durch Nutzerevaluation herausstellen.

Für eine gute Benutzbarkeit ist es auch wichtig, wie gut sich der Anwender im Fehlerfall informiert fühlt und wie sehr er den Eindruck hat, dass die Anwendung für die Lösung seines Problems geeignet ist. Dies sind subjektive, wenn auch teilweise messbare Kriterien, die im Rahmen einer Benutzer-Evaluation geprüft werden können, sobald die Anwendung genügend Funktionen unterstützt, die ausprobiert werden können. Dies wird erst nach Ende dieser Arbeit geschehen, weil der hier implementierte Funktionsumfang noch zu gering ist.

2.1.6 Effizienz

Die Effizienz ist für das Programm insofern wichtig, dass der Benutzer in einem angemessenen zeitlichen Rahmen mit einer Antwort rechnen darf (Relevanz: B).

Ob eine Anwendung effizient ist, richtet sich laut der DIN/ISO 9126 nach zwei Kriterien:

Zeitverhalten	Antwort- und Verarbeitungszeiten sowie Durchsatz bei der Funktionsausführung
Verbrauchsverhalten	Anzahl und Dauer der benötigten Betriebsmittel für die Erfüllung der Funktionen

Tabelle 5: Aspekte des Qualitätsmerkmals „Effizienz" (Starke, 2011)

Da es für beide Effizienz-Kriterien keine Zahlenvorgaben von Seiten des Unternehmens gibt, sie aber insgesamt für die Benutzer eine wichtige Rolle spielen, wird Effizienz nur als allgemeines Ideal im Rahmen der Implementierung berücksichtigt. Siehe hierzu auch *REQ-015*.

Eine Entscheidung, die sich positiv auf das Zeitverhalten auswirkt, wurde allerdings auf der Architekturseite getroffen: Das Modell wird nicht mehr vollständig in den Arbeitsspeicher geladen, wie es beim Straus API Tool der Fall war. Stattdessen wird nur der Teil, dessen Daten für die aktuelle Aufgabe notwendig sind, eingelesen und verarbeitet. So werden längere Lade- und Aktualisierungszeiten gespart.

2.1.7 Änderbarkeit

Unter dem Begriff „Erweiterbarkeit" wurde dieses Kriterium schon in der Aufgabenstellung als wichtig beurteilt (Relevanz: A). Als Aspekte des Qualitätsmerkmals „Änderbarkeit" schlägt die DIN/ISO 9126 Analysierbarkeit, Modifizierbarkeit, Stabilität und Prüfbarkeit vor. Die Begriffe werden in Tabelle 6 erklärt.

Analysierbarkeit	Aufwand, um Mängel oder Ursachen von Versagen zu Diagnostizieren oder um änderungsbedürftige Teile zu bestimmen
Modifizierbarkeit	Aufwand zur Ausführung von Verbesserungen, zur Fehlerbeseitigung oder Anpassung an Umgebungsänderungen
Stabilität	Wahrscheinlichkeit des Auftretens unerwarteter Wirkungen von Änderungen
Prüfbarkeit	Aufwand, der zur Prüfung der geänderten Software notwendig ist

Tabelle 6: Aspekte des Qualitätsmerkmals „Änderbarkeit" (Starke, 2011)

Um eine gute **Analysierbarkeit** zu erreichen, werden bei der Implementierung folgende Punkte berücksichtigt:

- Ausführliche Logging-Einträge (*REQ-006*)
- Konsequentes Dokumentieren des Codes (*REQ-008*)

- Großflächige Testabdeckung des Codes
- Geworfene Exceptions enthalten Informationen über die Situation, in der sie aufgetreten sind

Zu einer guten **Modifizierbarkeit** tragen bei:

- Schichtenarchitektur
- Planung in Komponenten
- Implementierung gegen Interfaces

Die **Stabilität** des Programms über die Zeit soll durch die Einplanung zukünftiger Anforderungen ermöglicht werden. Außerdem vereinfacht die modulare Bauweise ein Einfügen von beliebigen Funktionen, ohne dass die Übersicht der Oberfläche oder des Quellcodes leiden muss.

Die **Prüfbarkeit** der Software soll durch Unit-Tests sichergestellt werden. Unit-Tests bilden auch die Grundlage für jede Refaktorisierung der Software. Siehe dazu auch (Fowler, 2005) und Kapitel 2.3.

2.1.8 Übertragbarkeit

Als nächstes Qualitätskriterium nennt die DIN/ISO 9126 die Übertragbarkeit. Dazu werden als Unterpunkte Anpassbarkeit, Installierbarkeit und Konformität vorgeschlagen. Beschreibungen dieser Begriffe sind in Tabelle 7 angegeben.

Anpassbarkeit	Software an verschiedene festgelegte Umgebungen anpassen
Installierbarkeit	Aufwand, der zum Installieren der Software in einer festgelegten Umgebung nötig ist
Konformität	Grad, in dem die Software Normen oder Vereinbarungen zur Übertragbarkeit erfüllt

Tabelle 7: Aspekte des Qualitätsmerkmals „Übertragbarkeit“ (Starke, 2011)

Abgesehen von dem Aspekt **Installierbarkeit** ist die Übertragbarkeit für das Straus Multitool nicht relevant. Das Programm soll nur firmenintern verwendet werden. Dabei ist klar, dass auf den Arbeitsplatz-Rechnern ein Microsoft Windows-Betriebssystem laufen wird, aktuell Windows XP oder Windows 7. Damit ist die Lauffähigkeit von unter .NET entwickelten Programmen gesichert. Für das Straus Multitool ergibt sich damit insgesamt eine Relevanz auf B-Niveau.

Die Installierbarkeit ist ein wichtiger Aspekt (siehe auch *REQ-017*), gerade vor dem Hintergrund, dass das Straus API Tool hierbei Probleme bereitet. Nicht überall läuft das Programm beim ersten Versuch und es müssen vom Benutzer per

Hand Dateien kopiert werden. Dem soll beim Straus Multitool mit einer Installationsroutine begegnet werden, die automatisch alle notwendigen Pakete installiert, die das Programm verwendet und eine Ordnerstruktur erstellt, in der das Programm lauffähig ist. Hierfür gibt es von Microsoft verschiedene vorgefertigte Techniken. Dieser Aspekt muss für die Architektur also keine Berücksichtigung finden und ist in der Durchstich-Implementierung nicht vorgesehen.

2.1.9 Austauschbarkeit

Als letztes Kriterium nennt die DIN/ISO 9126 die Austauschbarkeit. Das Straus Multitool kann mit Hilfe einer Installationsroutine leicht anstelle des bisherigen Programms eingesetzt werden. Dafür muss nur das alte Programm gelöscht und das neue installiert werden. Hierfür sind keine Maßnahmen an der Architektur erforderlich. Damit wird die Relevanz dieses Kriteriums für die Entwicklung des Programms als gering (C) eingestuft.

2.2 Planung der Architektur

Um sich in der Wahl des Lösungsansatzes an bewährten Techniken zu orientieren wurde eine Literaturrecherche durchgeführt. Das Ergebnis war nicht so umfangreich wie erwartet: Es existieren wenige Architektur-Muster, es gibt viele Heuristiken zum Software-Entwurf und einige Vorgehensmodelle. In diesem Abschnitt werden die gefundenen Möglichkeiten kurz vorgestellt, um ein Bild vom Lösungsraum zu vermitteln.

2.2.1 Architekturmuster

Als Entwurfsmuster auf Architekturebene sind drei sehr bekannte Muster zu nennen: Layer-, Pipes&Filters- und Client-Server-Architekturen.

„All well structured object-oriented architectures have clearly-defined layers.“ – Mit diesem Satz spricht sich Grady Booch (Booch, 1996 S. 54) deutlich für die Verwendung von Schichten als Strukturierungselement einer Software aus. Vorteilhaft nach Gernot Starke (Starke, 2011) an der Verwendung von Schichten ist, dass diese voneinander unabhängig sind. Die Implementierung einer Schicht kann leicht durch eine andere Implementierung ausgetauscht werden, wenn in der Austausch-Schicht die gleichen Funktionen angeboten werden. Vor allem minimiert die Schichtenbildung Abhängigkeiten zwischen Komponenten und führt zu einem schnelleren Verständnis der Architektur.

Das Prinzip des Pipes&Filters-Musters ist die nacheinander erfolgende schrittweise Bearbeitung von Daten. Ein Beispiel für dieses Muster ist der Compiler, der den Quellcode schrittweise in verschiedenen Phasen analysiert. Dieses Muster eignet sich nicht für das Straus Multitool, weil es einen anderen Prozess vorschreibt, als von Overdick erwartet wird. Für eine Client-Server-Architektur ist die Aufgabenstellung ebenfalls wenig geeignet. Auch wenn die Anwendung dieses Architektur-Musters nicht unbedingt eine Verteilung auf mehrere Systeme bedeuten würde, so sind in der Aufgabenstellung auch keine Anhaltspunkte für regelmäßige Dienste zu finden.

2.2.2 Komponenten

Die Empfehlung, eine Software-Architektur als eine Struktur aus einzelnen Komponenten aufzubauen, findet sich häufig in der gängigen Literatur; vgl. (Microsoft Patterns & Practices Team, 2009), (Siedersleben, 2004), (Starke, 2011), (Szyperski, et al., 2002) und (Wirfs-Brock, et al., 2002). Dahinter steckt einerseits der Wunsch, die Aufgabe in immer kleinere Unteraufgaben zu zerteilen um sie möglichst leicht überblicken zu können. Andererseits erleichtert die Strukturierung der Objekte in Komponenten auch den Umgang mit Abhängigkeiten, indem Schnittstellen definiert werden, über die die Komponenten miteinander kommunizieren.

In dieser Arbeit soll die Definition von Johannes Siedersleben (Siedersleben, 2004 S. 42f) verwendet werden, in dessen Augen jede Komponente die folgenden Merkmale besäße:

1. Sie exportiert eine oder mehrere Schnittstellen[…].
2. Sie importiert andere Schnittstellen[…].
3. Sie versteckt die Implementierung und kann daher durch eine andere Komponente ersetzt werden, die dieselbe Schnittstelle exportiert[…].
4. Sie eignet sich als Einheit der Wiederverwendung, denn sie kennt nicht die Umgebung, in der sie läuft, sondern macht darüber nur minimale Annahmen […].
5. Komponenten können andere Komponenten enthalten[…].
6. Die Komponente ist neben der Schnittstelle die wesentliche Einheit des Entwurfs, der Implementierung und damit der Planung.

Das Ideal einer Komponente drückt sich durch geringe Kopplung und hohe Kohäsion aus. Das bedeutet, dass zwei Komponenten möglichst unabhängig voneinander sein sollten (geringe Kopplung), die Elemente innerhalb einer Komponente

aber logisch und inhaltlich eng zusammengehören sollten (hohe Kohäsion). Nach Helmut Balzert führt eine hohe Kohäsion zu spezialisierten Modulen, die besser wiederverwendbar und einfacher zu warten und zu ändern sind (Balzert, 1998).

2.2.3 Quasar

Der Name Quasar steht für „Qualitätssoftwarearchitektur" und beschreibt eine Sammlung von Heuristiken, die im Softwarehaus sd&m gesammelt wurden. In seinem Buch „Moderne Software-Architektur" beschreibt Johannes Siedersleben, wie man größere Systeme strukturieren und planen kann (Siedersleben, 2004). Aus diesem Buch werden einige Anregungen und Definitionen verwendet. Aufgrund der verhältnismäßig geringen Größe des Straus Multitools wurde eine vollständige Entwicklung nach Quasar als unverhältnismäßig beurteilt und ausgeschlossen.

2.3 Die Komponenten

Für die Entwicklung einer neuen Architektur ist es zunächst wichtig, die Verantwortlichkeiten des Systems so auf Bausteine zu verteilen, dass die Abhängigkeiten minimiert werden. Je weniger Abhängigkeiten in einem System bestehen, desto leichter ist es zu verstehen. Auch Änderungen können leichter realisiert werden, da Bausteine einfach ausgetauscht oder hinzugefügt werden können.

Die Bausteine der höchsten Ebene sind die Komponenten. Um eine möglichst geringe Abhängigkeit zu erreichen, werden Klassen zu Komponenten zusammengefasst, die ihrerseits Schichten zugeordnet werden

Zur Einteilung der Architektur in Schichten, dient der Standard-Vorschlag für Softwareschichten von Gernot Starke als Orientierung (siehe auch (Starke, 2011 S. 143ff)). Dieser Vorschlag sieht das Vorhandensein der folgenden Schichten vor:

- Präsentation
- Applikationsschicht (optional)
- Fachdomäne
- Infrastruktur

Für die neue Architektur wurden die drei nicht-optionalen Schichten Präsentation, Fachdomäne und Infrastruktur übernommen. Der Vorteil des Schichtenmodells ist unter anderem in der Vorschrift begründet, dass möglichst nur von höheren Schichten auf niedrigere Schichten zugegriffen wird. Dadurch werden zyklische Abhängigkeiten vermieden und das System bleibt flexibel.

Der Standard-Vorschlag für Software-Schichten von Gernot Starke beinhaltet folgende Beschreibungen zu den einzelnen Schichten: Die **Präsentationsschicht** enthält nur Funktionen der Benutzeroberfläche. Damit kann bei Bedarf die konkrete Benutzeroberfläche leicht ausgetauscht werden. Die **Fachdomäne** soll den Entwicklern ermöglichen, sich auf die fachlichen Aspekte des Systems zu konzentrieren, während die **Infrastrukturschicht** die Komplexität der technischen Infrastruktur kapselt. Zur technischen Infrastruktur gehören:

- Persistenz und Datenhaltung
- Physikalische Infrastruktur (Schnittstellen zur Hardware)
- Integration von Fremdsystemen
- Sicherheit
- Kommunikation und Verteilung

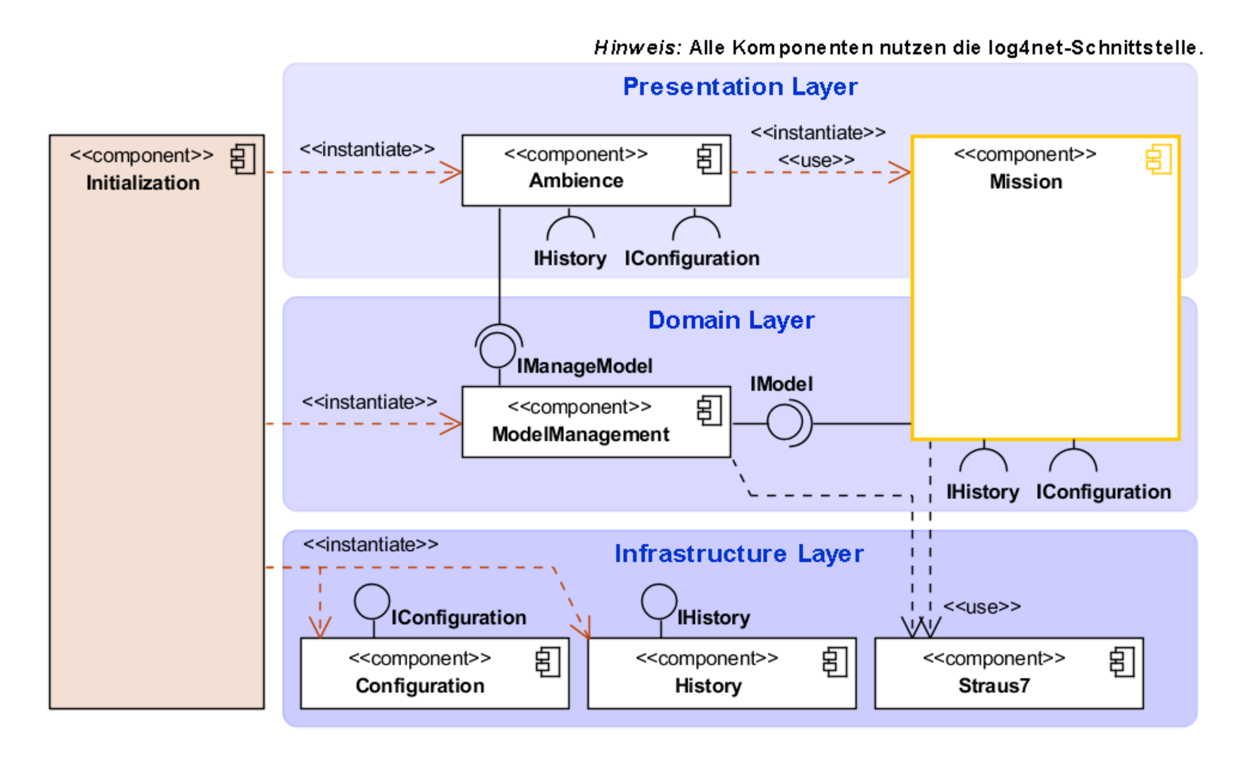


Abbildung 4: Die Architektur des Straus Multitools

Nach Gernot Starke und Peter Hruschka entstehen sinnvolle Architekturen nur in iterativen Entwurfsprozessen (Starke, et al., 2011). Es wird also ein Entwurf für eine Architektur ausgearbeitet, dieser Entwurf wird geprüft, per Durchstich umgesetzt und aus den so gewonnenen Erfahrungen entsteht ein neuer Entwurf. Diesen Vorgang, bei dem die interne Struktur der Software geändert wird, ohne ihr beobachtetes Verhalten zu ändern, nennt man Refaktorisierung; vgl. (Fowler, 2005).

Auch zwischenzeitlich veränderte Anforderungen können in den neuen Entwurf aufgenommen werden, dann spricht man allerdings nicht mehr von einer Refaktorisierung. Nach gründlicher Beschäftigung mit den Anforderungen und mit iterativer Vorgehensweise ist die Architektur in Abbildung 4 entstanden.

Die verschiedenen Schichten, Komponenten und Schnittstellen sollen im Folgenden aus der Sicht von außen, als sogenannte „Black Box" beschrieben werden. Die Zuordnung der Komponenten zu den genannten Schichten in dem Abschnitt zu der jeweiligen Komponente erläutert.

2.3.1 CONFIGURATION

Die `Configuration`-Komponente sorgt dafür, dass Daten über das Programmende hinaus gespeichert werden können und damit beim nächsten Programmstart wieder zur Verfügung stehen. Sie erfüllt also die Forderung nach Persistenz und ist somit der Infrastruktur-Schicht zugeordnet.

Zum Speichern müssen die Daten als Schlüssel-Wert-Paare vorliegen: Jeweils ein Schlüssel zum Wiederfinden der Information und ein Wert, der unter diesem Schlüssel abgelegt werden soll, bilden ein Paar. Die Schnittstelle `IConfiguration` bietet im Wesentlichen zwei Methoden an:

```
string getValue(string key);
void changeValue(string key, string newValue);
```

Damit kann von außen ein Wert gelesen oder geändert werden. Das Hinzufügen von neuen Wertepaaren über Programmcode ist nicht vorgesehen, da während der Laufzeit des Programms keine neuen Paare entstehen.

2.3.2 HISTORY

Die `History`-Komponente ermöglicht generell Undo- und Redo-Funktionen (*REQ-057*). Da diese Funktion die Sicherheit der Daten zur Programmlaufzeit unterstützt, ist die `History`-Komponente ebenfalls der Infrastruktur-Schicht zugeordnet.

Für die Umsetzung der Komponente soll das Command-Muster (Gamma, et al.) verwendet werden. Dieses Muster beinhaltet, dass ein Kommando als Objekt modelliert wird. Damit besteht die Möglichkeit dem Kommando-Objekt Informationen mitzugeben, wie das Kommando rückgängig gemacht werden kann; vgl. (Freeman, et al., 2005)

Für die `History`-Komponente werden zwei Schnittstellen benötigt: Die Schnittstelle, die beschreibt, wie ein Kommando auszusehen hat und die Schnittstelle für die

Benutzung der `History`-Komponente von außen. Aufgrund einer Doppeldeutigkeit des Begriffs „Command“ in .NET wird ein Kommando in der Implementierung immer mit „Instruction“ bezeichnet.

Das `IInstruction`-Interface schreibt folgende Methoden vor:

```
void execute();
bool undo();
bool redo();
```

Damit kann jede Instruction, die das Interface implementiert, ausgeführt werden und anschließend rückgängig gemacht und wieder hergestellt werden.

Die Verwaltung darüber, welche Instruction aktuell diejenige ist, die rückgängig gemacht werden soll, wenn der Benutzer den Undo-Befehl auslöst, übernimmt die `History`-Komponente. Angesprochen werden die Funktionen über das Interface `IHistory`. Die Methoden, die mit der Nutzung dieser Schnittstelle den äußeren Komponenten zugesichert werden, sind:

```
bool undo();
bool redo();
void done(IInstruction instruction);
```

Sie stellen die Möglichkeiten her, dass eine Instruction sich als ausgeführt anmelden kann (`done`), rückgängig gemacht werden kann (`undo`) und anschließend erneut ausgeführt werden kann (`redo`).

2.3.3 Straus7

Die Straus-API-Funktionen werden im Straus Multitool nicht direkt angesprochen. Es ist eine Komponente für den API-Zugriff vorgesehen, die die passenden Methodensignaturen und weitere Hilfsfunktionen enthält. Das liegt daran, dass die API mit dem .NET-Framework nur korrekt in VisualBasic.NET angesprochen werden kann und nicht mit C#. Deshalb wird diese Komponente in VisualBasic.NET entwickelt und stellt die Funktionen der Straus-API für das restliche Programm bereit (*REQ-023*). Da es sich bei der Aufgabe dieser Komponente um die Integration eines Fremdsystems handelt, wird die Komponente der Infrastruktur-Schicht zugeordnet.

2.3.4 Mission

Die `Mission`-Komponente ist in Abbildung 4 gelb umrandet und hebt sich somit von den anderen Komponenten ab. Das soll bedeuten, dass es von dieser Komponente verschiedene Implementierungen geben wird. Jede Aufgabe, die in einem geschlossenen Kontext gelöst werden kann, wird in einer eigenen `Mission`-

Komponente implementiert. Für den Durchstich soll hier stellvertretend für alle noch kommenden `Mission`-Komponenten die Berechnung der Windlasten (*REQ-022* und *REQ-022a-c*) implementiert werden.

Jedes `Mission`-Modul hat seine eigenen Oberflächenelemente, die im Rahmen des Hauptprogramms angezeigt werden, sodass die Funktionen dem Benutzer zur Verfügung stehen, sobald diese `Mission`-Komponente instanziiert und geladen wurde.

2.3.5 ModelManagement

Die `ModelManagement`-Komponente ist für das Öffnen, Schließen und Speichern des Straus-Modells zuständig (*REQ-026* und *REQ-050*). Sie greift dazu auf die `Straus7`-Komponente zu und nutzt für diese Zwecke bereitgestellte Funktionen. Pro Instanz der `ModelManagement`-Komponente kann nur ein Modell gleichzeitig geöffnet sein. Da sich die Komponente ausschließlich mit der Fachdomäne beschäftigt, wird sie in dieser Schicht eingeordnet.

Es stehen zwei verschiedene Schnittstellen zur Verfügung, die beide von der Komponente implementiert werden, um für verschiedene Perspektiven jeweils den optimalen Zugriff zur Verfügung zu stellen. Von der Oberfläche her (`Ambience`-Komponente, siehe Abschnitt 2.3.6) sollen Öffnen, Speichern und Schließen unterstützt werden. Dies geschieht über das Interface `IModelManager`, das die folgenden Methoden und Felder vorschreibt:

```
int id { get; }              // read-only

bool opened { get; }         // read-only
bool modified { get; }       // read-only
bool apiInitialized { get; }// read-only

bool openModel(string filename);
bool saveModel();
bool saveModelAs(string filename);
bool closeModel();
```

Das Feld `id` enthält die eindeutige Nummer, über die die Straus-API das Modell identifiziert. Wenn die API initialisiert ist, enthält das Feld `apiInitialized` den Wert `true`. Über die Felder `opened` und `modified` kann der Zustand des verwalteten Modells ausgelesen werden, während die Funktionen `openModel()`, `saveModel()`, `saveModelAs(string filename)` und `closeModel()` Befehle der Oberfläche umsetzen.

Für eine `Mission`-Komponente ist nur wichtig, mit welchem Modell gearbeitet werden soll und wie dessen Zustand ist. Deshalb kann es das Interface `IModel` nutzen, über das nur drei Zustandsfelder zur Verfügung stehen:

```
int id { get; }
bool modified { set; get; }
bool apiInitialized { get; }
```

Hier ist das Feld `modified` allerdings auch schreibbar, damit die Mission-Komponente den Zustand aktualisieren kann, wenn sie das Modell seit dem Öffnen verändert hat.

2.3.6 Ambience

Die `Ambience`-Komponente enthält einen Großteil der Oberfläche des Programms mit dessen Funktionen. Deshalb ist sie in der Präsentationsschicht angeordnet. Sie definiert das Hauptmenü und belegt es mit Funktionalität. Außerdem instanziiert sie mit Öffnen eines Modells alle passenden `Mission`-Komponenten und ordnet deren Steuerelemente in die Oberfläche ein. Die Sprache der Oberfläche ist Englisch (*REQ-056*).

Im Folgenden wird eine Übersicht über die beim Start vorhandenen Buttons[1] und ihre Aufgaben gegeben. Dazu gehört auch, an welche Komponenten die Aufgaben delegiert werden.



Eine Straus-Modelldatei öffnen.
→ `ModelManagement`-Komponente aufrufen, `Mission`-Komponenten initialisieren



Die geöffnete Straus-Modelldatei speichern (*REQ-026*).
→ `ModelManagement`-Komponente aufrufen



Die geöffnete Straus-Modelldatei unter neuem Namen speichern.
→ `ModelManagement`-Komponente aufrufen



Die geöffnete Straus-Modelldatei schließen (*REQ-050*), nicht speichern.
→ `ModelManagement`-Komponente aufrufen



Die letzte Aktion zurücknehmen (Undo, *REQ-057*).
→ `History`-Komponente aufrufen

---

[1] Die verwendeten Icons sind im Internet frei verfügbar und stammen aus folgenden Quellen:

- Oxygen Icon Set, `www.oxygen-icons.org`
- Tango Icons, `http://commons.wikimedia.org/wiki/Tango_icons`



Die zurückgenommene Aktion erneut durchführen (Redo, *REQ-057*).
→ `History`-Komponente aufrufen



Ein Straus-Modellfenster öffnen (*REQ-014*).
→ `Straus7`-Komponente benutzen



Ein Fenster öffnen, in dem die programmweiten Einstellungen verändert werden können.
→ `Configuration`-Komponente aufrufen



Das Benutzerhandbuch anzeigen (*REQ-003* und *REQ-009*).

In Abbildung 5 ist die Programmoberfläche vollständig abgebildet. Zur Zeit der Aufnahme ist ein Modell geöffnet. Die `Windloads`-Komponente ist geladen und deren Oberflächenelemente werden in einer Registerkarte dargestellt.

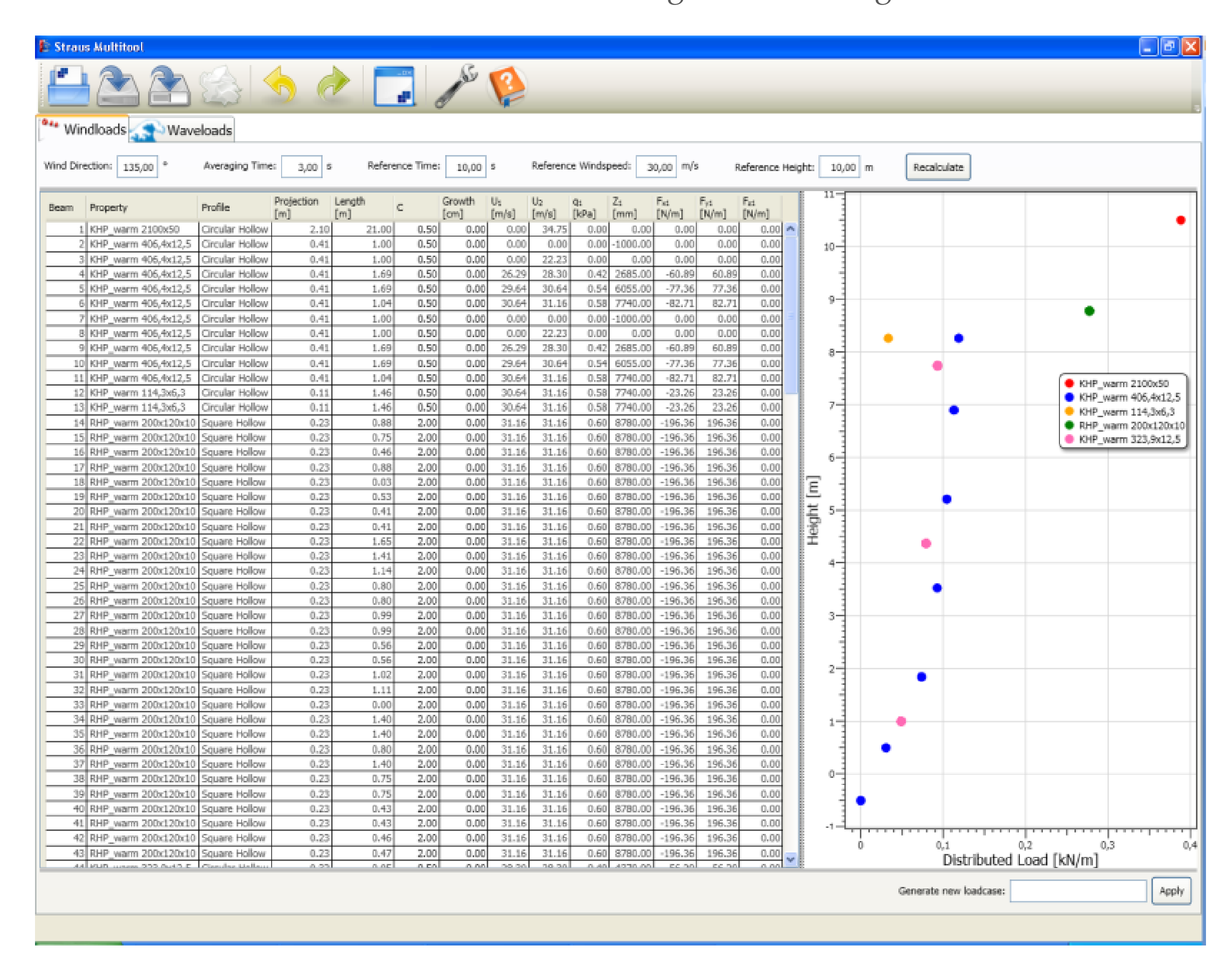


Abbildung 5: Grafische Benutzeroberfläche des Straus Multitools

2.3.7 Initialization

Die `Initialization`-Komponente übernimmt die Herstellung des „Settings". Sie instanziiert alle Komponenten (mit Ausnahme von `Mission`-Komponenten) und übergibt ihnen jeweils Referenzen auf andere Komponenten, die sie benötigen. Abbildung 6 zeigt den Startbildschirm, der während der Arbeitszeit der `Initialization`-Komponente sichtbar ist. Im Anschluss startet sie die `Ambience`-Komponente.



Abbildung 6: Startbildschirm des Straus Multitools

3 Beschreibung der Komponenten

In diesem Kapitel wird die Innenansicht der einzelnen Komponenten vorgestellt. Dabei enthält jeder Abschnitt die Beschreibung einer Komponente und folgt dabei diesem Aufbau: Zunächst wird der Zweck der Komponente zusammengefasst. Anschließend werden die Vor- und Nachbedingungen der Komponente beschrieben sowie ihre innere Struktur. Dann folgt eine Beschreibung von Maßnahmen zur Testbarkeit, sowie wichtige getroffene Designentscheidungen und deren Begründung.

3.1 Configuration-Komponente

Die `Configuration`-Komponente verwaltet global die Konfiguration des Programms. Es können Paare von Zeichenketten übergeben, wieder abgerufen und persistent abgelegt werden.

3.1.1 Vor- und Nachbedingungen

Die Komponente erwartet eine bestehende XML-Datei mit der in Abbildung 7 gezeigten Struktur. Bei der Instantiierung der Komponente muss der Pfad zu dieser Datei angegeben werden. Alle Schlüssel-Wert-Paare, die mit der `Configuration`-Komponente verarbeitet werden sollen, müssen in der XML-Datei beim Starten des Programms schon angelegt sein.

```xml
<?xml version="1.0" encoding="utf-8"?>
<Configuration>
  <Item>
    <Key>scratchpath</Key>
    <Value>C:\Multitool\Data\ModelScratchFolder\</Value>
  </Item>
  <Item>
    <Key>gravity</Key>
    <Value>-z</Value>
  </Item>
</Configuration>
```

Abbildung 7: Die Datei configuration.xml

Die XML-Datei wird beim Beenden des Programms in dem Zustand hinterlassen, der als letztes im Programm hergestellt wurde. Falls Werte verändert wurden, sind diese gespeichert.

Die in Abbildung 7 gezeigten Schlüssel-Wert-Paare werden in dieser Form vom Straus Multitool verwendet. Abgesehen von „gravity“ und „scratchpath“ gibt es

aktuell noch keine weiteren Schlüssel in der Konfigurationsdatei. Dies wird sich aber mit der Erweiterung des Programms ändern, wenn die Einstellungen für hinzukommende Module angelegt werden.

3.1.2 Struktur

Die innere Struktur der `Configuration`-Komponente ist in Abbildung 8 als UML-Schema dargestellt. Die Komponente setzt sich im Wesentlichen aus drei Klassen zusammen: Der Klasse `Configuration`, die die Schnittstelle zu anderen Modulen implementiert, die Klasse `XmlConfigurationManager`, die sich um die Verbindung mit der XML-Datei kümmert, diese liest schreibt und interpretiert, und die Klasse `ConfigurationItem`, mit der ein Schlüssel-Wert-Paar modelliert ist. Diese drei Klassen sind der funktionale Kern der Komponente. Eine vierte Klasse, `WrongFormatException`, die von `Exception` erbt, dient der passenden Benachrichtigung des Anwenders, wenn die XML-Datei syntaktisch oder semantisch nicht konform ist.

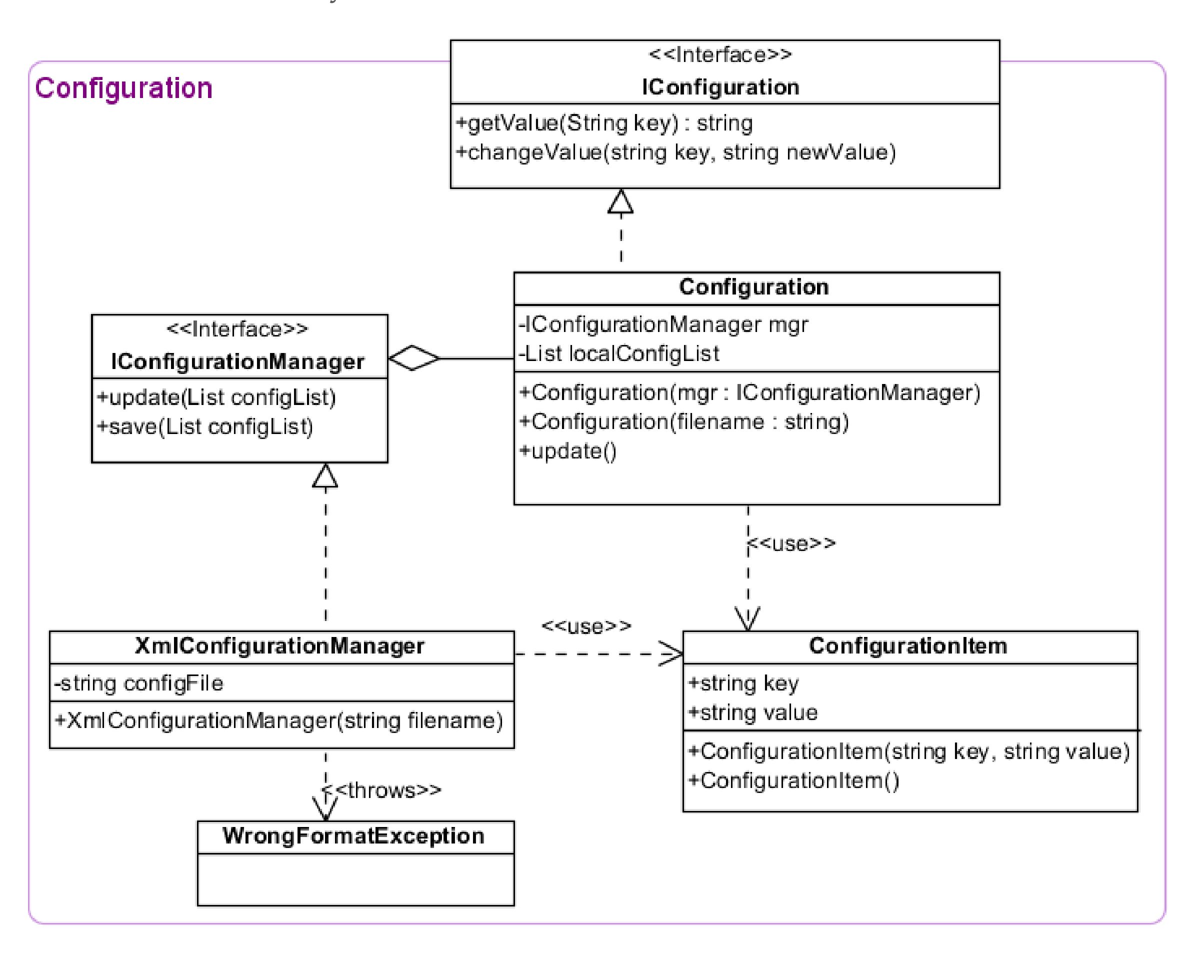


Abbildung 8: Die Configuration-Komponente von innen

Die Schnittstelle nach außen wird durch das Interface `IConfiguration` definiert. Dies sollte in allen Komponenten, die die `Configuration`-Komponente nutzen, verwendet werden, damit die Komponente im Bedarfsfall leicht ausgetauscht werden kann.

Dies ist zum Beispiel bei Unit-Tests interessant, aber auch, wenn das Konzept der `Configuration`-Komponente geändert werden soll.

Die Klasse `WrongFormatException` erbt ihre Eigenschaften von der Klasse `Exception`. Dieser Umstand wurde der Übersicht halber in der Darstellung weggelassen. Die Schnittstelle `IConfigurationManager` dient der Testbarkeit der Klasse. Mehr dazu im folgenden Abschnitt.

3.1.3 Testbarkeit

Mit herkömmlichen Methoden von Unit-Tests lässt sich lediglich die `Configuration`-Klasse testen, weil sich diese einfach isolieren lässt. Die Klasse `XmlConfigurationManager` greift unmittelbar auf das Dateisystem zu und lässt sich somit nicht so einfach vom Umfeld trennen. Um die Klasse dennoch zu testen, kann ein Mock-Framework wie Moles, vgl. (Microsoft Research), verwendet werden, das in der Lage ist, auch statische Klassen und Funktionen aus .NET-Bibliotheken im Testfall zu ersetzen. Viele Open-Source Mock-Frameworks, wie Rhino Mocks (Hibernating Rhinos LTD), bieten diese Möglichkeit nicht an. Eine weitere Möglichkeit, die Klasse `XmlConfigurationManager` zu testen, sind Integrationstests, die das Zusammenspiel von Klassen untersuchen.

In diesem Abschnitt wird zunächst gezeigt, wie die `Configuration`-Klasse getestet werden kann. Anschließend werden mögliche Integrationstests für die Klasse `XmlConfigurationManager` thematisiert.

Um die `Configuration`-Klasse zu testen, wird diese isoliert und von Testklassen umgeben (Abbildung 9, grün/fett umrandete Klassen). Die Klasse `ConfigurationItem` kann dabei ohne Änderungen weiter verwendet werden, weil sie keine Logik beinhaltet, sondern lediglich als Container dient. Sollte sich dies ändern, muss der Test mit einem Stub für die Klasse `ConfigurationItem` durchgeführt werden.

Für die Tests wird ein `StubConfigurationManager` implementiert, der so konfigurierbar ist, dass er die Funktionen des `XmlConfigurationManagers` nachahmen kann und sich jeweils so verhalten kann, wie der Test es fordert. Auf diese Weise kann die Klasse `Configuration` mit beliebigen Umgebungsszenarios getestet werden.

Für jeden Test wird in der Klasse `ConfigurationTests` eine Instanz der Klasse `Configuration` und ein `StubConfigurationManager` erzeugt. Es wird jeweils eine bestimmte Umgebungssituation geschaffen, anschließend wird die zu testende Funktion ausgeführt bevor der Erfolg dieser Funktion mit dem NUnit-Testframework überprüft wird, vgl. (Poole, et al.).

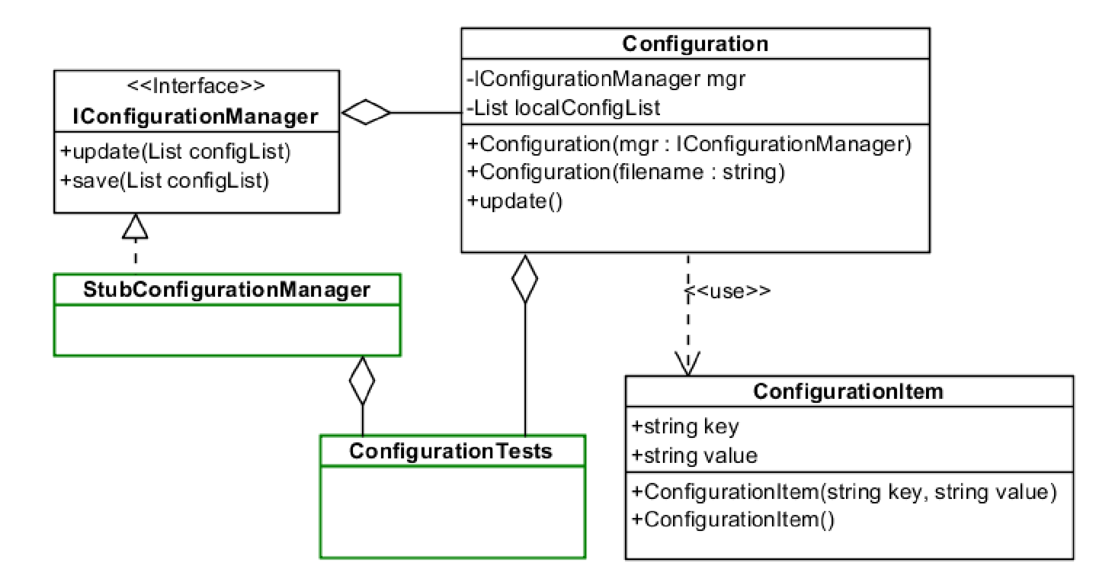


Abbildung 9: Teststruktur der Configuration-Komponente

Eine Reihe von Tests, die die innere Logik der Komponente abdecken sollen, sind mit Hilfe eines `StubConfigurationManagers` bereits implementiert worden. Weitere mögliche Testfälle, die die Komponente als Blackbox betrachten, sich also nur die äußere Schnittstelle beziehen, sind in Tabelle 8 aufgeführt. Diese Testfälle beziehen sich auf den Zustand der XML-Datei, sind also Integrationstests. Hier kann die Klasse `XMLConfigurationManager` nicht ersetzt werden, sondern ist im Gegenteil wichtigster Gegenstand der Tests. Diese Testfälle lassen sich mit Hilfe des NUnit-Testframeworks implementieren, sind aber streng genommen keine Unit-Tests, da mehrere Klassen gleichzeitig getestet werden und dabei unmittelbar auf eine Datei zugegriffen wird.

Testfall	Arrangieren	Agieren	Bestätigen
1	Gültige XML-Datei ohne Einträge	`getValue(key)`	`noSuchKeyException`
2	Ungültige XML-Datei	`getValue(key)`	Exception
3	Keine XML-Datei	`getValue(key)`	Exception
4	Gültiges Key-Value-Paar in XML-Datei	`getValue(key)`	Value wird zurück gegeben
5	Gültige XML-Datei mit Key-Value-Paaren	`getValue(notExistingKey)`	`noSuchKeyException`
6	Gültige XML-Datei mit einem Paar, Value zu lang für den Datentyp String	`getValue(key)`	Exception
7	Gültige XML-Datei ohne Einträge	`changeValue(key, value)`	`noSuchKeyException`

8	Ungültige XML-Datei	`changeValue(key, value)`	Exception
9	Keine XML-Datei	`changeValue(key, value)`	`IOException`
10	Gültige XML-Datei mit Key-Value-Paaren	`changeValue(key, value)`	`getValue(key)` gibt value zurück
11	Gültige XML-Datei	`changeValue(notExistingKey, value)`	`noSuchKeyException`

Tabelle 8: Mögliche Testfälle für die Configuraion-Komponente

Eine Umsetzung der genannten Testfälle mit dem NUnit-Framework wird hier umrissen: In der Tabelle sind XML-Dateien mit unterschiedlichem Inhalt gefordert. Dazu sollten im Test-Ordner unterschiedliche Dateien mit dem entsprechenden Inhalt angelegt werden, die für den jeweiligen Testfall genutzt werden. Über die `TearDown`-Methode des Testframeworks muss nach jedem Test der ursprüngliche Zustand der Dateien wieder hergestellt werden. Dazu sollte keine Methode der Komponente verwendet werden, weil diese ja noch getestet werden soll. Besser wäre es, Prototyp-Dateien zu halten, mit denen jeweils die veränderte Datei wieder überschrieben wird.

3.1.4 Designentscheidungen

Eine Reihe von Möglichkeiten zur Gestaltung dieser Komponente wurde im Laufe der Konzipierung evaluiert. Diese Entscheidungen sollen im Folgenden kurz begründet werden.

Kein dynamisches Hinzufügen und Löschen von Schlüssel-Wert-Paaren über die Komponenten-Schnittstelle. Einstellungen, die langfristig gespeichert werden sollen, stehen schon bei der Entwicklung des Programms fest. Es gibt bisher keinen Grund, warum zur Laufzeit des Programms Werte hinzugefügt werden sollten. Deshalb wurde beschlossen, dass der Entwickler die bestehende XML-Datei um weitere Paare zu ergänzen muss, wenn er diese benutzen möchte.

Schlüssel nicht als XML-Tag modelliert. In der Konfigurationsdatei hätte der Schlüssel, z. B. „gravity“ auch direkt als XML-Tag modelliert werden können. Also `<gravity>-z</gravity>` anstelle von `<Item><Key>gravity</Key><Value>-z</Value> </Item>`. Die gewählte Variante erlaubt aber, die Items um weitere Werte zu erweitern. Zum Beispiel um eine textuelle Beschreibung hinzuzufügen, die dynamisch in die Oberfläche des Konfigurationsfensters eingebunden werden kann. Auch für die Umsetzung der Anforderung *REQ-007*, die das Speichern einer Menge von zusammenhängenden Werten als Parameter-Konfiguration fordert, ist diese Notation sinnvoll. Mit der gewählten Variante steht auch einer möglichen Einführung der dynamischen Moduleinbindung nichts im Weg. Die `Configuration-`

Komponente müsste nur um einige Funktionen erweitert werden, die es ermöglichen, die Konfigurationsdatei um neue Einträge zu ergänzen und die komplexeren Daten zurückzugeben.

Nur `string` als Datenformat für Keys und Values. Die Möglichkeit, die Schlüssel-Wert-Paare in verschiedenen Zahlenformaten zu speichern, hätte die Schnittstelle der Komponente auf ein Vielfaches vergrößert. Außerdem werden die Werte in der Datei als Text abgelegt, was eine umständliche Umwandlung der Werte innerhalb der Komponente bedeutet hätte. Das Format, in dem ein Schlüssel-Wert-Paar gespeichert werden soll, wäre immer eingeschränkt durch die von `IConfiguration` vorgegebenen Datentypen. Werden aber alle Schlüssel und Werte einheitlich als Strings behandelt, kann die Umwandlung und Interpretation der Werte außerhalb der Komponente erfolgen und jeder Nutzer der Configuration-Komponente kann sich sein eigenes Zahlenformat definieren und den erhaltenen String selbst interpretieren.

Einführung der `IConfigurationManager`-Schnittstelle. Diese Schnittstelle wurde eingefügt, damit die Klasse `Configuration` mit Hilfe von Unit-Tests getestet werden kann. So kann der Teil der Komponente, der für den Dateizugriff zuständig ist, durch einen Stub ersetzt werden, damit die `Configuration`-Klasse isoliert ist.

Keine Verwendung des Singleton-Musters. Das Singleton-Muster scheint sich gerade im Infrastructure-Layer anzubieten: Komponenten, die eindeutig und einzigartig sein sollen und die von nahezu allen anderen Komponenten benutzt werden, sind perfekte Kandidaten für dieses Muster. Generell spricht aber gegen Singletons, dass sie zum Testen nicht durch Fake-Objekte ersetzt werden können. Sie können also nicht ausgetauscht werden, was ein sinnvolles Unit-Testen (ohne den Einsatz eines Mock-Frameworks) unmöglich macht. Diese Entscheidung gilt für alle Komponenten im Infrastructure-Layer.

Kein Default-Konstruktor für Configuration. Ein parameterloser Konstruktor für die Configuration-Klasse würde bedeuten, dass es einen Standard-Pfad für die Konfigurationsdatei geben müsste, der in diesem Fall von der Klasse genutzt wird. Würde eine solche Einstellung umgesetzt werden, würde dies aber bedeuten, dass von mehreren Stellen auf die gleiche Datei geschrieben wird, falls mehrere Instanzen der Komponente erstellt werden. Damit könnte die Konsistenz der Datei nicht mehr garantiert werden.

3.2 History-Komponente

Die `History`-Komponente verwaltet eine globale Undo- und eine globale Redo-Liste (*REQ-057*), auf denen die ausgeführten Kommandos abgelegt werden. Bei Bedarf wird ein Kommando zurückgenommen oder erneut ausgeführt. Zusätzlich stellt die `History`-Komponente das Interface `IInstruction` bereit, nach dessen Vorgabe alle Kommandos implementiert werden müssen, die invertiert werden können.

3.2.1 Vor- und Nachbedingungen

Zur Benutzung der `History`-Komponente sind keine Vor- oder Nachbedingungen definiert. Mit der Initialisierung eines `History`-Objekts kann dieses über das Interface `IHistory` von beliebig vielen Objekten genutzt werden, die Kommandos ausführen. Diese Kommandos müssen lediglich das `IInstruction`-Interface implementieren.

3.2.2 Struktur

Die `History`-Komponente besteht aus einem `History`-Objekt und zwei Interfaces (siehe Abbildung 10). Das `History`-Objekt übernimmt die Verwaltung des Undo- und des Redo-Stacks. Objekte, die das Interface `IInstruction` implementieren, können sich über die `IHistory`-Schnittstelle bei dem `History`-Objekt per `done`-Methode registrieren und werden somit auf dem Undo-Stack abgelegt. Zum detaillierten Vorgehen siehe (Krypczyk, 2011).

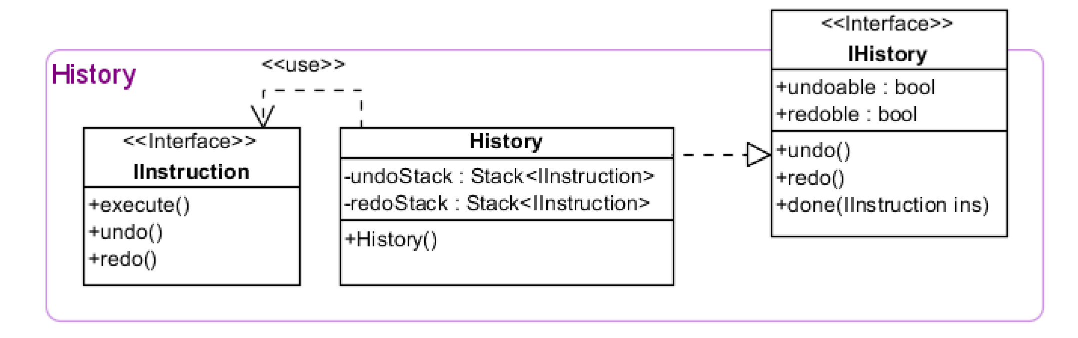


Abbildung 10: Klassendiagramm der History-Komponente

3.2.3 Testbarkeit

Um die Komponente testen zu können, wird ein Objekt der Klasse `History` erstellt und eine Dummy-Instruction implementiert. Diese ist von der `HistoryTests`-Klasse völlig kontrollierbar und hilft, das Verhalten der `History`-Klasse zu untersuchen. Die vollständige Teststruktur ist in Abbildung 11 dargestellt. Im Vergleich zu Abbildung 10 ist zu erkennen, dass zwei Methoden zur `History`-Klasse hinzugefügt

worden sind. Sie sind in Abbildung 11 grün markiert. Über diese beiden Methoden ist die Tiefe des Undo- und des Redo-Stacks auslesbar. Darüber kann das Verhalten der `History`-Klasse ebenfalls überprüft werden.

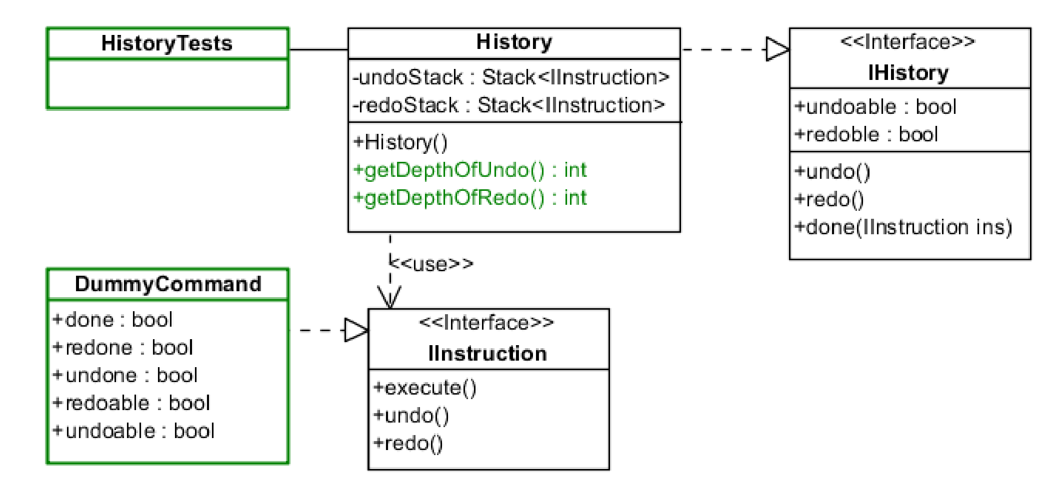


Abbildung 11: Teststruktur der History-Komponente

Auch für die `History`-Komponente wurden Testfälle erdacht, die in Tabelle 9 aufgeführt werden. Die Auflistung beschreibt Testfälle, die aus der Sicht von außen auf die Komponente, also der Blackbox-Sicht, entstanden sind.

Testfall	Arrangieren	Agieren	Bestätigen
1	Leerer Undo-Stack	`undo()`	Exception
2	Leerer Undo-Stack		`undoable` ist `false`
3	`DummyCommand` ausführen	`undo()`	`DummyCommand.undone` ist `true`
4	Leerer Redo-Stack	`redo()`	Exception
5	Leerer Redo-Stack		`redoable` ist `false`
6	`DummyCommand` ausführen, `undo()` ausführen	`redo()`	`DummyCommand.redone` ist `true`

Tabelle 9: Mögliche Testfälle für die History-Komponente aus Blackbox-Sicht

Der letzte Testfall in Tabelle 9 lässt sich zum Beispiel auch leicht zu einem Test für die Robustheit erweitern. Bei einem einzigen Element auf dem Stack müssten die Flags `undoable` und `redoable` abwechselnd `true` sein, wenn `undo()` und `redo()` im Wechsel ausgeführt werden.

3.2.4 Designentscheidungen

Die Entscheidung für die Herangehensweise mit dem Command-Muster und den Stacks soll in diesem Abschnitt begründet werden. Für das Zurücknehmen von Aktionen gibt es im Allgemeinen mehrere Lösungen, die im Folgenden kurz angerissen werden sollen.

Checkpoints: Basiert auf der Idee, einen Schnappschuss vom aktuellen Zustand der Applikation zu machen, um diesen später wieder herstellen zu können (Wikipedia). Dies würde für das hier behandelte Programm bedeuten, dass die Straus-Modelldatei mehrfach gespeichert werden müsste. Da diese Dateien häufig Größen im Gigabyte-Bereich erreichen, wurde diese Technik als Lösung verworfen. Das Zwischenspeichern würde zu lange dauern und der Speicherplatzbedarf wäre zu hoch.

Speichern von Differenzen: Man könnte die Änderungen als Differenzen zwischen Dateien speichern, wie es in Versionskontrollsystemen, wie Subversion, gehandhabt wird. Allerdings konnte keine Bibliothek gefunden werden, die ein solches Verfahren unterstützt und schon ausreichend getestet ist. Subversion bietet direkt eine Programmierschnittstelle an, vgl. (Collins-Sussman, et al., 2009 S. 190ff). Da sich die Differenzen auf eine Modelldatei beziehen, wäre auch CVS eine denkbare Option. Auch andere Versionskontrollsysteme wie git, mercurial usw. könnten auf entsprechende Möglichkeiten untersucht werden. Bisher wurden diese Ideen aber wegen zu hohen Aufwands verworfen. Auch wird befürchtet, dass sich ein Commit nach jeder Aktion des Straus Multitools zu nachteilig auf die Performance auswirken könnte.

Transactions: Im Namespace `System.Transaction` von .NET findet sich die Möglichkeit, Transaktionen zu definieren, die mittels Rollback wieder zurückgenommen werden können. Diese Möglichkeit wurde verworfen, weil sich die Bibliothek auf Datenbanken bezieht und die Struktur der Dateien so beibehalten werden soll. Eine zusätzliche Einführung einer Datenbank zum Speichern der Zwischenstände wurde wegen zu hohen Aufwands verworfen.

Stacks / Command-Muster: Diese Möglichkeit ist verhältnismäßig einfach zu implementieren und bietet nebenbei die schnelle Erweiterbarkeit um das Aufzeichnen von Makros, was seitens Overdick als mögliche Erweiterung genannt wurde (*REQ-018*). Außerdem kann so entschieden werden, welche Aktionen als invertierbar implementiert werden sollen und bei welchen dies nicht wichtig ist.

Ein weiterer Vorteil bei der Benutzung des Command-Musters ist: Wenn das Straus Multitool durch das Aufnehmen von Makros erweitert werden soll (*REQ-018*), kann dies leicht durch Anpassung der `History`-Komponente geschehen. Denn sie bietet durch das `IInstruction`-Interface schon die Möglichkeit an, Kommandos zu wiederholen.

3.3 Straus7-Komponente

Die Straus7-Komponente dient der Anbindung des Programms an die Straus-API. Dadurch unterscheidet sie sich stark von den anderen Komponenten, weil einige Vorgaben der Straus-Entwickler eingehalten werden müssen. Zuerst einmal ist die Programmiersprache VisualBasic.NET, weil die API nach Auskunft der Entwickler nicht korrekt in C# eingebunden werden kann. Zum anderen unterscheidet sich ihre Struktur stark von der anderer Komponenten, siehe hierzu auch Abschnitt 3.3.2 „Struktur“.

3.3.1 Vor- und Nachbedingungen

Um die Straus-API verwenden zu können, muss die Straus7-Software auf dem Arbeitsplatzrechner des Benutzers installiert sein und über eine gültige Lizenz verfügen. Die Installation von Straus soll Vorbedingung der Installation des neuen Programms sein, die gültige Lizenz muss bei jedem Programmstart überprüft werden, weil sich die Lizenz auf einem USB-Stick befindet. Ist dieser nicht eingesteckt, funktioniert die Straus-API nicht und das Straus Multitool kann nicht sinnvoll arbeiten. Die Überprüfung wird in der `Initialization`-Komponente übernommen. Anschließend ruft die `ModelManagement`-Komponente die Initialisierungsfunktion der Straus-API auf, die vor der Verwendung aller anderen API-Funktionen aufgerufen werden muss, bevor eine Straus-Datei geöffnet wird. Vorher dürfen keine Benutzerfunktionen aktiviert sein, die die Straus-API verwenden, weil sich diese sonst in einem undefinierten Zustand befindet.

Nach der Verwendung der `Straus7`-Komponente darf keine Straus-Datei mehr geöffnet sein und die Schnittstelle muss mit `St7Release()` freigegeben worden sein. Dies geschieht beim Schließen des Straus Multitools.

3.3.2 Struktur

Wie bereits angedeutet, unterscheidet sich die Struktur der `Straus7`-Komponente deutlich von der der anderen Komponenten. Nach Außen weist sie kein Interface für die Kommunikation mit anderen Komponenten auf. Intern besteht sie aus drei `Modules` und einer Klasse. `Modules` sind eine Eigenheit der Programmiersprache VisualBasic.NET. Sie enthalten Methoden und Variablen, die global zugreifbar sind, wenn das `Module` nicht als `private` deklariert wurde.

Die Klasse definiert eine spezielle Exception, die `ApiException`, die im ganzen Programm für Ausnahmen verwendet wird, die durch Funktionen der Straus-API auftreten. Das Modul `Errors` enthält nur die Funktion `handleError()`, die in der Lage

ist, die Fehlercodes der API in eine aussagekräftigere Zeichenkette umzuwandeln. Sie verwendet dazu eine Funktion der Straus-API. Darüber hinaus enthält die Komponente die beiden `Modules St7APICall` und `St7APIConst`, die die Funktionsköpfe und die Konstanten enthalten, die von den Straus7-Entwicklern zur Nutzung der API zur Verfügung gestellt werden. Die Dateien werden mit der Straus7-Installation ausgeliefert und unverändert verwendet. Durch die Verwendung dieser beiden `Modules` kann nun auch von anderen Komponenten aus über die `Straus7`-Komponente auf die API zugegriffen werden.

3.3.3 Testbarkeit

Diese Komponente ist nicht auf einem herkömmlichen Weg mit Unit-Tests überprüfbar. Die Fehlerstring-Methode greift direkt auf die API-Funktionen zu. Sie könnte mit dem Moles-Framework (Microsoft Research) getestet werden. Die beiden anderen `Modules` gehören direkt zur API, deshalb werden hier keine Tests angesetzt. Die Klasse `ApiException` enthält keine Logik und muss deshalb nicht getestet werden.

Die Struktur der `Straus7`-Komponente beeinflusst aber ganz entscheidend die Testbarkeit aller Objekte, die über sie auf die Straus-API zugreifen. Deshalb werden im nachfolgenden Abschnitt Designentscheidungen dokumentiert, die innerhalb der `Straus7`-Komponente getroffen werden mussten, die aber auch alle anderen Komponenten auf der Domänenschicht betreffen.

3.3.4 Designentscheidungen

Es ist wichtig für die Testbarkeit der Komponenten, die die `Straus7`-Komponente verwenden, dass sie vom direkten Zugriff auf das Fremdsystem isoliert werden können. Dies spräche für die Einführung eines `Interfaces`, das den Zugriff auf die Straus-API beschreibt. Dies ist jedoch nicht möglich, weil die Dateien, die für die Nutzung der API eingebunden wurden, `Modules` enthalten und diese keine `Interfaces` implementieren können. Da `Modules` auch weder instanziiert noch beerbt werden können, entfällt damit auch die nächste Möglichkeit für das Isolieren der Komponente: Die Methode „Extract and Override“ nach Roy Osherove (Osherove, 2010), bei der innerhalb des Tests die abzukapselnde Klasse beerbt und die Methoden überschrieben werden.

Es besteht die Möglichkeit, eine komplette Abstraktionsschicht einzuführen, die die API vollständig kapselt und deren Funktionen überschreibbar sind. Da es aber über tausend Funktionen sind und deren Benutzung für die Entwicklung späterer

`Mission`-Komponenten nicht eingeschränkt werden soll, wurde diese Vorgehensweise wegen zu hohen Aufwands verworfen.

Im Sinne der Testbarkeit wurde die Entscheidung getroffen, „Extract und Override" bei Funktionen anzuwenden, die die Straus-API benutzen. Deshalb müssen die Methoden innerhalb der Klassen so aufgeteilt werden, dass die Methoden, die auf die API zugreifen, keine zu testende Logik enthalten. Damit liegt die Nahtstelle für die Tests nicht zwischen den Komponenten, sondern innerhalb der Klassen, die auf die Straus-API zugreifen. Eine Nahtstelle, auch Seam genannt, ist nach Michael Feathers „eine Stelle, an der Sie Verhalten in Ihrem Programm modifizieren können, ohne es an dieser Stelle zu ändern." (Feathers, 2011)

Die Entscheidung bringt den Nachteil mit sich, dass sie von der Implementierung nicht erzwungen wird. Sich an das Konzept zu halten erfordert sehr viel Disziplin vom Programmierer. Er muss jedes Mal selbst prüfen, ob er die wichtigen Methoden als `virtual` (überschreibbar) markiert hat und ob deren Umfang klein genug ist, um nicht getestet werden zu müssen.

3.4 Mission-Komponente

Die `Mission`-Komponente existiert in mehreren Ausprägungen und dient jeweils der Ausführung einer bestimmten Aufgabe. Damit bilden die `Mission`-Komponenten den wesentlichen Teil des Programms. Eine `Mission`-Komponente arbeitet sowohl auf der Präsentations- als auch auf der Domänenschicht und nutzt die Komponenten der Infrastrukturschicht. Die Einbindung in das Hauptprogramm erfolgt über die Präsentationsschicht durch die `Ambience`-Komponente.

Für die Durchstich-Implementierung wird eine Komponente ausgewählt, die Windlasten berechnet und in der Modelldatei speichert (*REQ-022* und *REQ-022c*). Die folgenden Betrachtungen gehen also von der `Windloads-Mission` aus. Aufgaben für weitere `Mission`-Komponenten können Anhang A entnommen werden, z. B. fordert *REQ-045* die Generierung von Wellenlasten.

3.4.1 Vor- und Nachbedingungen

Die `Windloads-Mission` muss in eine grafische Benutzeroberfläche eingebettet werden. Sie ist als grafisches `UserControl` implementiert, das eine Registerkarte (Tab) realisiert. Bevor die Registerkarte geladen wird, muss ein Modell zum Arbeiten geöffnet worden sein. Die Nachbedingungen richten sich nach den Handlungen

des Benutzers. Je nach Nutzung der Komponente kann die Modelldatei verändert sein oder auch nicht.

3.4.2 Struktur

Die Struktur der `Windloads`-Komponente richtet sich nach Strukturen innerhalb der Benutzeroberfläche. Ein Ausschnitt ist in Abbildung 12 dargestellt. Die Einbindungsstelle in das Hauptfenster ist für ein `Mission`-Modul ein grafisches `TabItem`, in dem die Ansicht des `WindloadsTab` angezeigt wird. `WindloadsTab` ist vom Typ `UserControl`, also ein benutzerdefinierte Oberflächenelement.

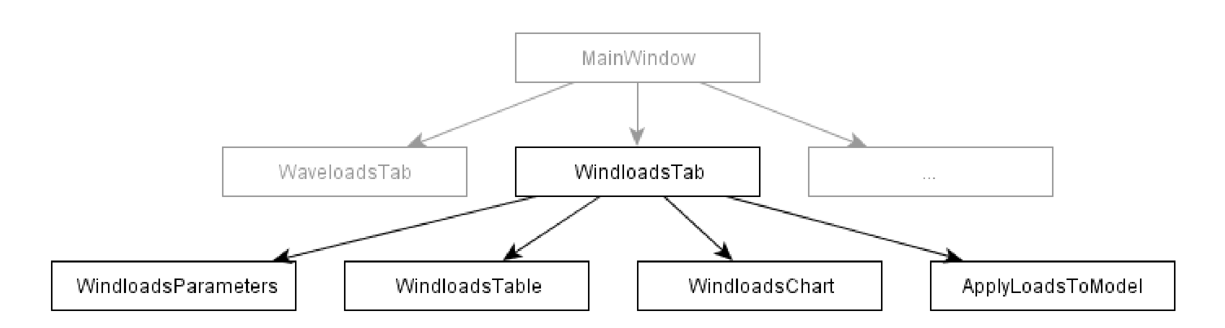


Abbildung 12: Strukturelle Anordnung der Oberflächenelemente

Das `UserControl WindloadsTab` unterteilt sich in vier weitere `UserControls`, die eine Funktionsgruppe bilden und sich auch optisch zusammenhängen. Es handelt sich hierbei um `WindloadsParameters`, `WindloadsTable`, `WindloadsChart` und `ApplyLoadsToModel`. Die Zuordnung der Oberflächenelemente zu den `UserControls` ist in Abbildung 13 dargestellt.

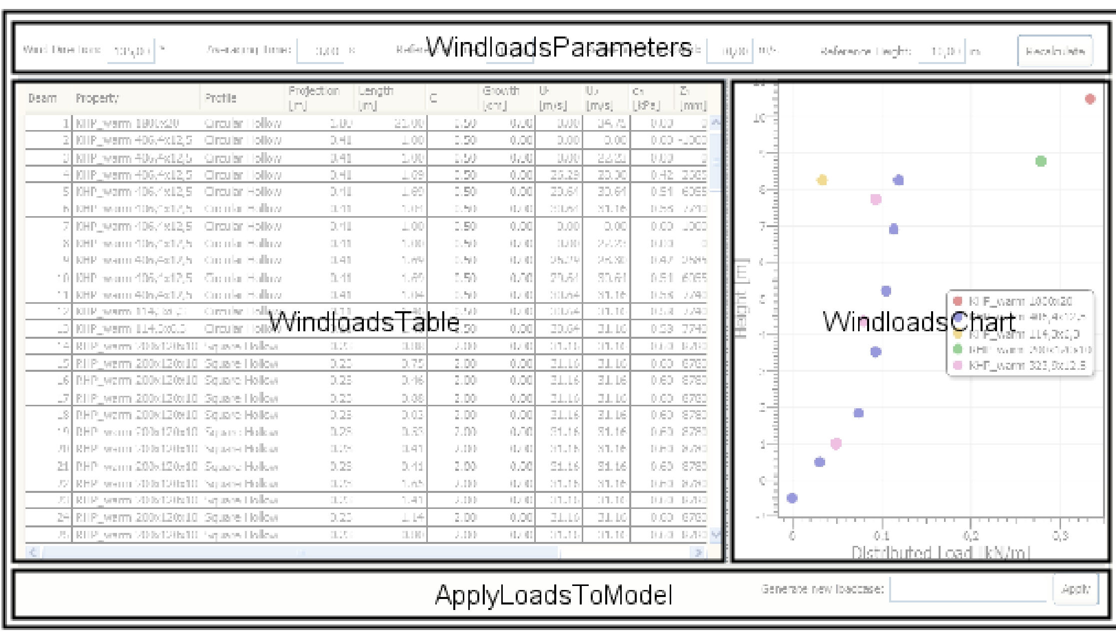


Abbildung 13: Unterteilung des Windloads-Tabs in einzelne UserControls

Mit dem `UserControl WindloadsParameters` können globale Parameter für die Windlastenberechnung eingestellt werden. `WindloadsTable` zeigt für jeden Stab (engl. „beam“) des geöffneten Modells spezifische Werte an. Dazu gehören Informationen über das Querschnittsprofil, die Länge, die Position im Raum und die Kräfte, die unter den aktuell eingestellten Bedingungen auf den Stab einwirken. Über das `WindloadsChart` werden die Kräfte abhängig von der Höhe aufgetragen. Dabei ergibt sich für jede Querschnittsart eine neue Datenreihe. Der Benutzer kann die berechneten Lasten in der Modelldatei über das `UserControl ApplyLoadsToModel` speichern.

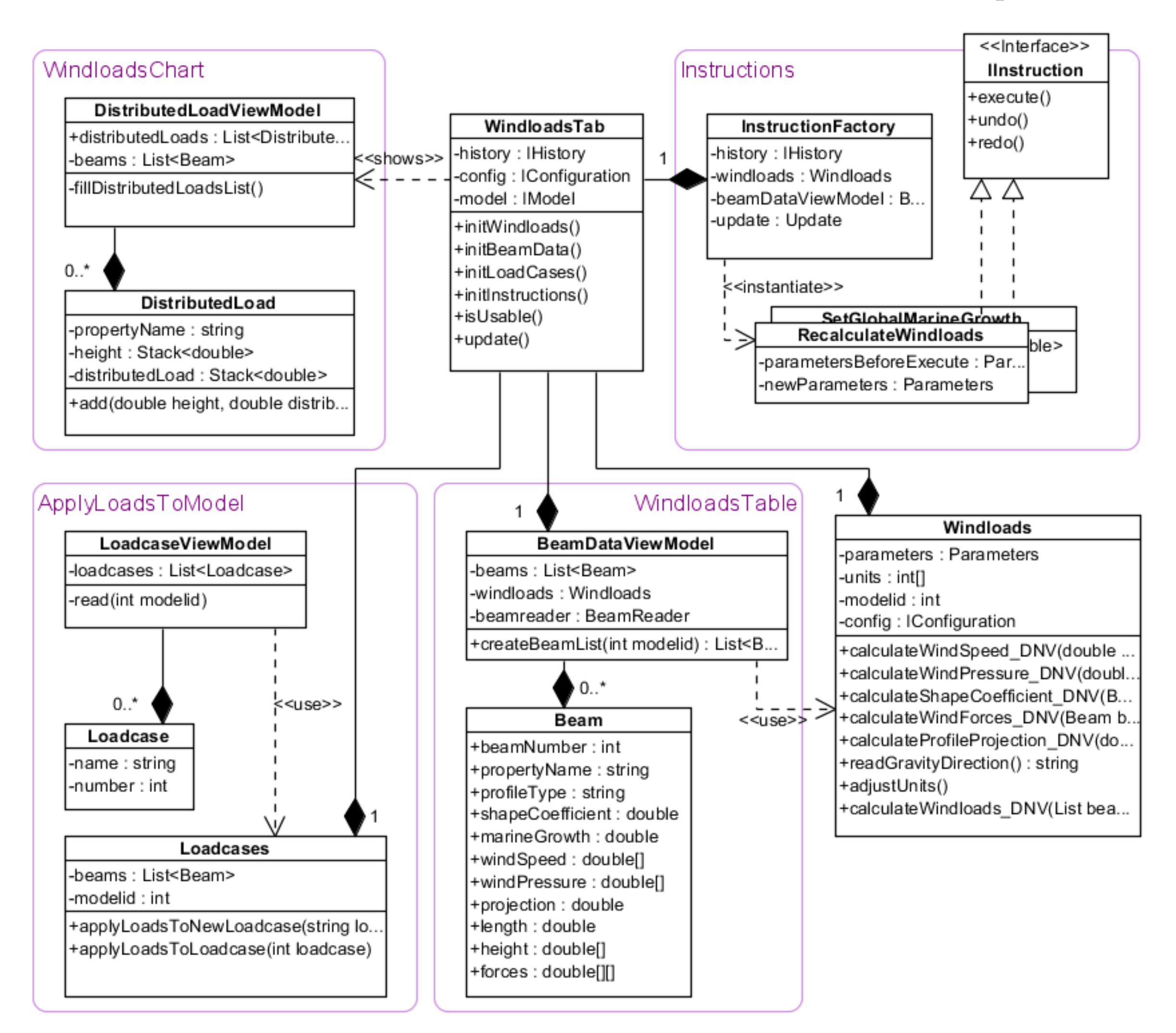


Abbildung 14: Klassendiagramm der Windloads-Mission

Die hier benannten Strukturen finden sich auch im Klassendiagramm der Komponente wieder. Die wesentlichen Elemente sind in Abbildung 14 dargestellt. Zentraler Ausgangspunkt der Struktur ist auch hier `WindloadsTab`. Die Klasse enthält Referenzen auf alle Einstiegspunkte in tiefere Ebenen. Im Wesentlichen sind auch die anderen Bereiche der Komponente nach `UserControls` aufgeteilt. Eine Ausnahme bilden die Bereiche `Instructions` und `Windloads`. Mit der `InstructionFactory` wurde

für alle Elemente eine zentrale Stelle geschaffen, an der zurücknehmbare Kommandos instanziiert werden. Damit bietet dieser Bereich einen Querschnittsservice, der den anderen Bereichen zur Verfügung steht. Der `Windloads`-Bereich berechnet die aus den eingestellten Parametern resultierenden Windlasten. Damit wird er ebenfalls von verschiedenen Stellen benutzt.

Im den folgenden Abschnitten sollen die fünf `UserControls` der `Windloads`-Komponente detaillierter beschrieben werden. Dabei werden Elemente aus Abbildung 14 wieder verwendet, präzisiert und teilweise in anderem Kontext dargestellt. Für das bessere Verständnis folgt zunächst ein Exkurs zur Oberflächentechnik WPF im .NET-Framework.

Exkurs

Die *Windows Presentation Foundation (WPF)* ist eine Technik zur Beschreibung und Programmierung von Oberflächen im .NET-Framework. Bei dieser Technik existieren für jede Oberflächen-Komponente eine *XAML-Datei* und eine Datei in einer Programmiersprache, z.B. in C#. Die XAML-Datei beschreibt allgemein Formateigenschaften wie Farbe, Größe und Position der Elemente, kann aber z.B. auch statische Einstellungen zur *Bindung* an Code-Elemente beinhalten. In der Datei mit dem Programmcode, der sogenannten *Code-Behind-Datei*, können beispielsweise Ereignishandler implementiert werden, die über die Bindung zur XAML-Datei bei bestimmten Oberflächenaktionen ausgelöst werden.

3.4.2.1 WindloadsTab

In der Code-Behind-Datei zum `WindloadsTab` werden die einzelnen Teile der Komponente initialisiert. Dazu werden Instanzen von benötigten Klassen erzeugt und diese werden miteinander verknüpft. Dieser Vorgang ist in Abbildung 15 abgebildet. Sämtliche in der Abbildung dargestellten Pins sind Input-Pins. Der Übersicht halber wurde weitestgehend auf die Darstellung des Objektflusses (schwarze Pfeile) verzichtet. Die roten Pfeile bilden den Kontrollfluss ab. Die unterschiedlich umrandeten Aktivitäten sollen die Initialisierungsschritte der grafischen Elemente (blau, fett) von denen für die Initialisierung der domainspezifischen Klassen abheben (schwarz).

Im Wesentlichen geschieht Folgendes: Zunächst werden die Objekte initialisiert, die im Hintergrund arbeiten und ihre Referenzen werden gehalten. Anschließend werden die vier `UserControls` initialisiert, aus denen `WindloadsTab` besteht. Ihnen werden dazu Referenzen auf Objekte mitgegeben, mit denen sie später kommunizieren müssen.

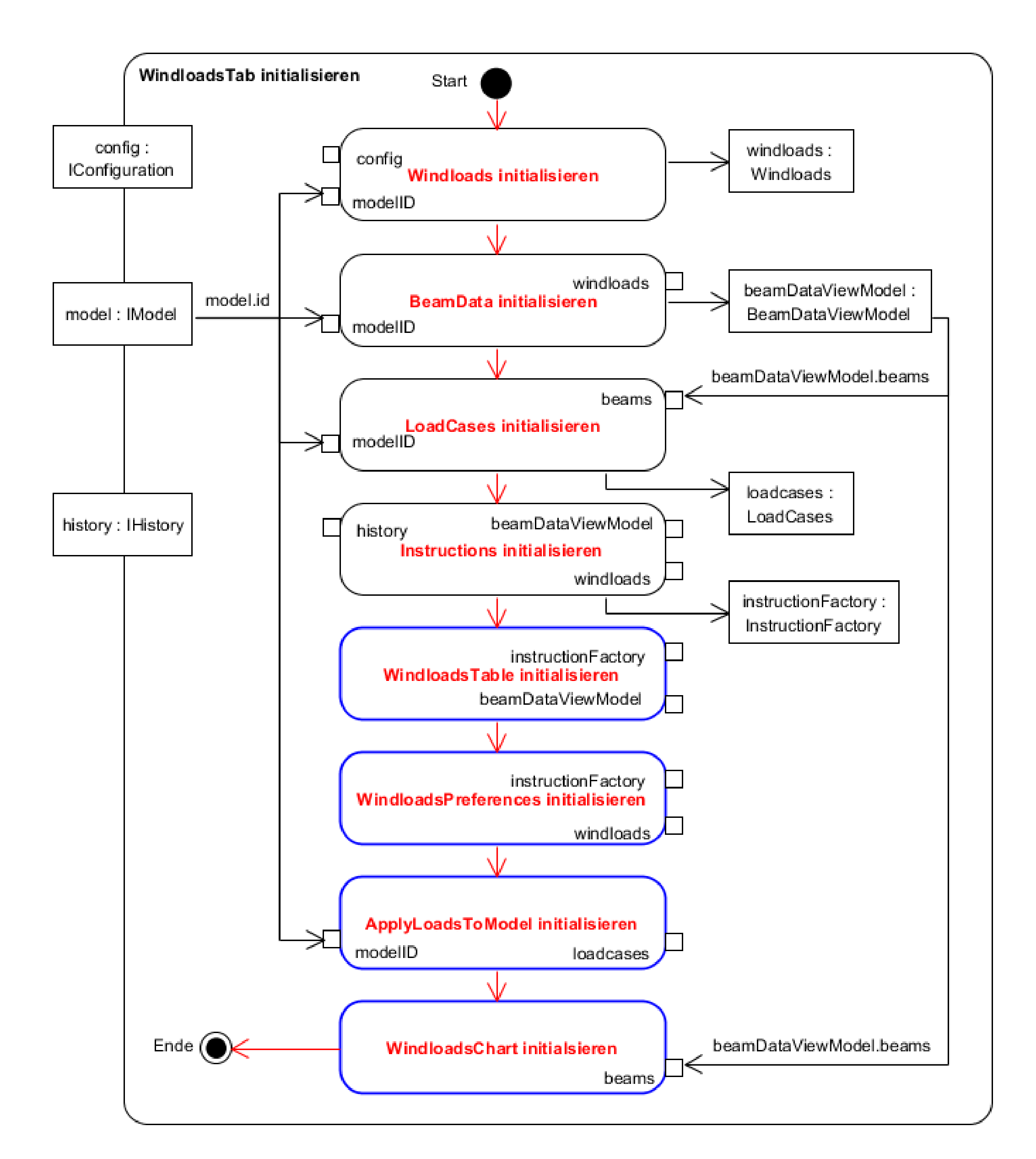


Abbildung 15: Initialisierung der Windloads-Komponente

3.4.2.2 WindloadsParameters

Das `UserControl WindloadsParameters` wird als Leiste von Eingabefeldern im oberen Teil des `WindloadsTabs` angezeigt. Hier kann der Nutzer Parameter einstellen, die bei der Windlastenberechnung berücksichtigt werden sollen, siehe auch *REQ-022*. Die innere Struktur der Code-Behind-Dateien für die `WindloadsParameters` ist in Abbildung 16 dargestellt.

Die Abbildung zeigt, dass das `UserControl WindloadsParameters` das `Parameters`-Objekt verändert, auf das die `Windloads`-Klasse bei der Berechnung zugreift. Um eine erneute Berechnung anzustoßen, ist die Instruction `RecalculateWindloads` implementiert worden, von der das `InstructionFactory`-Objekt eine Instanz herstellt, wenn es durch das `UserControl` dazu aufgefordert wird. Das `RecalculateWindloads`-Objekt

meldet sich nach der Ausführung bei der `History`-Komponente an. Damit lässt sich die Neuberechnung auch rückgängig machen, wobei die alten Parameter wieder eingestellt werden und die Rechnung erneut durchgeführt wird.

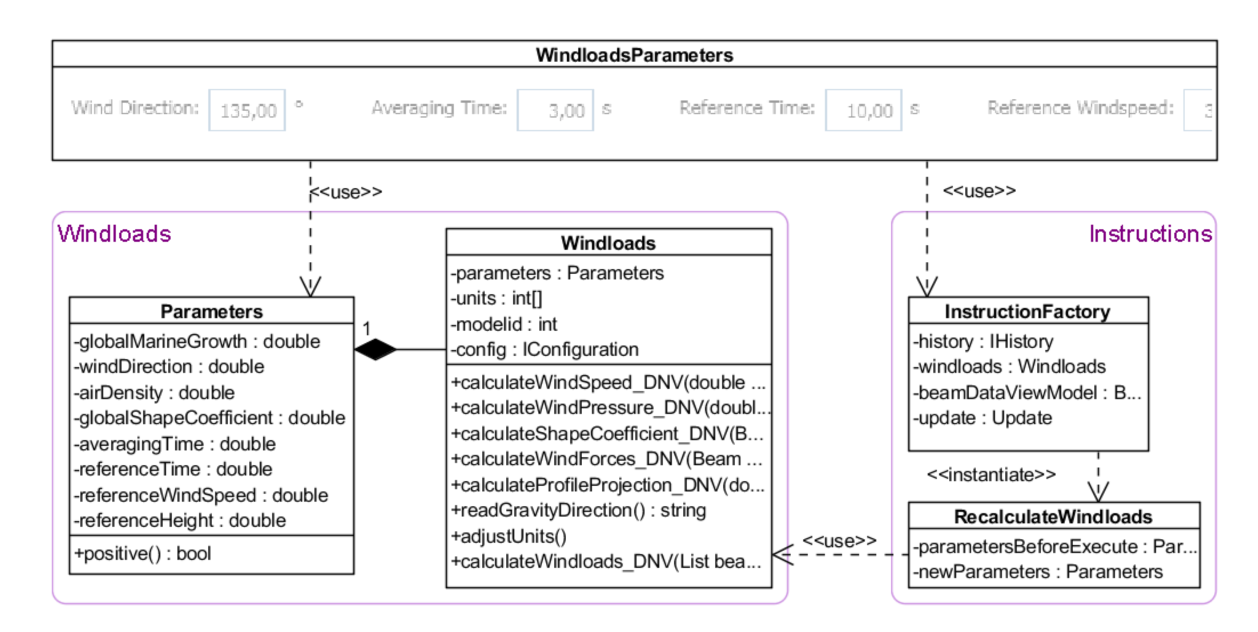


Abbildung 16: Klassendiagramm hinter den Parametern für die Windlasten

Zur leichteren Handhabung von Referenzen wurde die `InstructionFactory` eingeführt. Von ihr wird bei der Initialisierung der Komponente ein Objekt erstellt, das die notwendigen Referenzen zur `History`-Komponente und zum internen Modell enthält. Damit kann z. B. das `WindloadsParameters-UserControl` beliebig viele Instructions vom Typ `RecalculateWindloads` erstellen, die ausgeführt und rückgängig gemacht werden können. Das gleiche `InstructionFactory`-Objekt verwaltet auch Instructions für andere `UserControls`. Dieses Modell kann auch in anderen Mission-Komponenten verwendet werden, um die Ausführung von Instructions zu vereinfachen.

3.4.2.3 WindloadsTable

Das `UserControl WindloadsTable` hat zwei wichtige Funktionen. Zunächst einmal dient es der Darstellung der Eigenschaften der zur Konstruktion des Modells verwendeten Stäbe. Jede Zeile der Tabelle stellt die Stab-Nummer (Spalte „Beam“) zusammen mit einer Menge anderer Werte dar. Dazu gehören verschiedene Abmessungswerte, der angenommene maritime Bewuchs („Marine Growth“) an der Oberfläche des Stabs, die angreifenden Windgeschwindigkeiten an den beiden Endpunkten und die daraus resultierenden Kräfte (*REQ-022b*). Die Windgeschwindigkeit und die angreifenden Kräfte werden aus den anderen Werten berechnet(*REQ-022*).

Die zweite Funktion ist die Veränderung von Parametern. So können die Werte für „Marine Growth“ und den „ShapeCoefficient“ C für jeden einzelnen Stab in der Tabelle verändert werden, was eine Neuberechnung der Lasten anstößt. Die Parameter können entweder für die gesamte Spalte gleich oder für jede Zelle einzeln verändert werden. Diese Änderungen können rückgängig gemacht werden, da sie über Instructions realisiert werden, die sich nach Ausführung bei der `History`-Komponente registrieren.

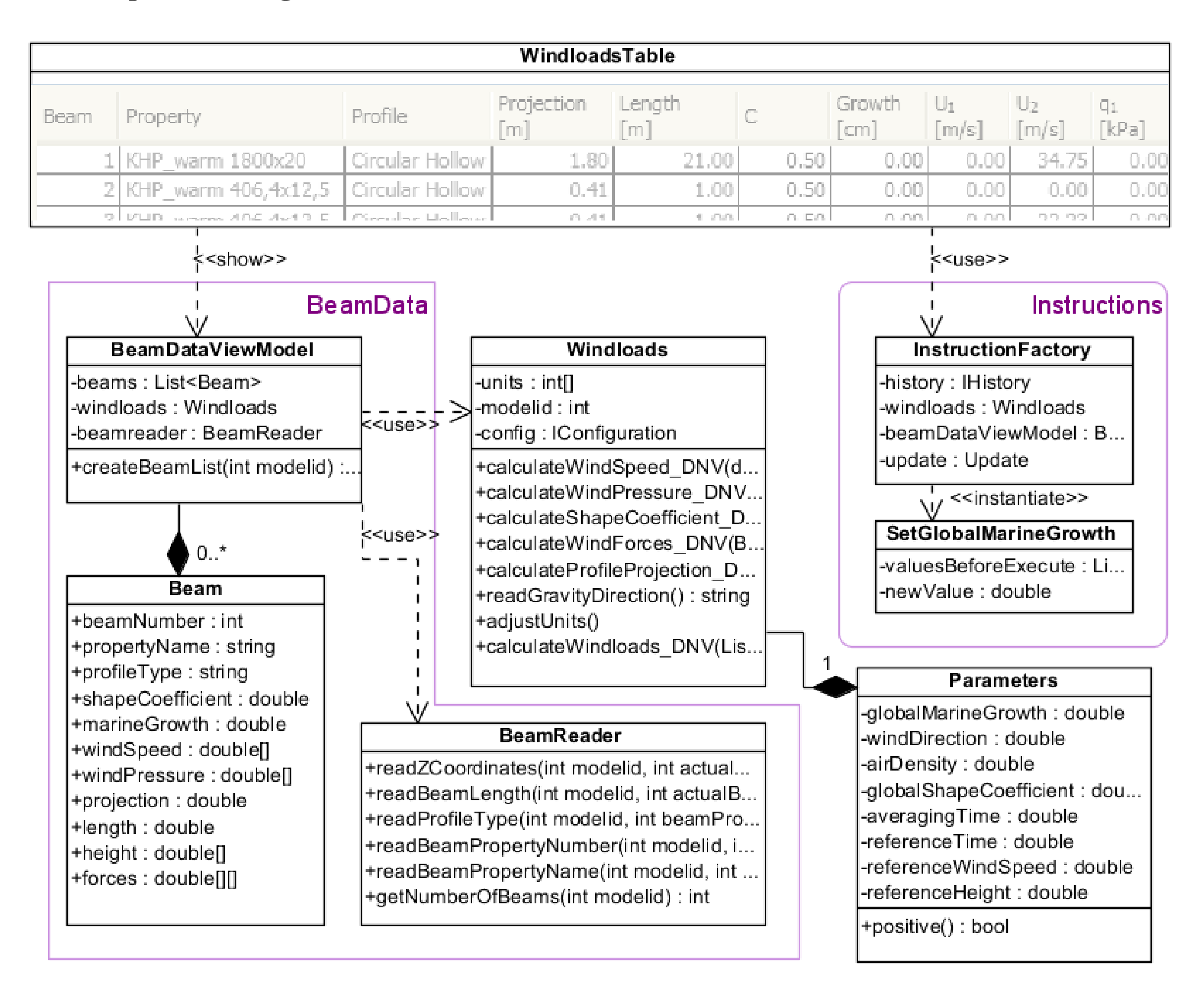


Abbildung 17: Klassendiagramm hinter dem UserControl WindloadsTable

Die Klassenstruktur hinter dem `UserControl WindloadsTable` ist in Abbildung 17 dargestellt. Links im Bild findet sich die Subkomponente `BeamData`, die das ViewModel für die Tabelle enthält. Mit der Initialisierung des `BeamDataViewModel`-Objekts werden die Stab-Daten aus dem Straus-Modell gelesen (*REQ-019*) und die erste Berechnung der Lasten über die `Windloads`-Klasse wird angestoßen (*REQ-022*). Rechts oben im Bild findet sich auch hier das `InstructionFactory`-Objekt, das hier die Instruction `SetGlobalMarineGrowth` erstellt. Damit kann der Wert für den maritimen Bewuchs für alle Stäbe gleichzeitig verändert werden. Falls der Benutzer vorher

aufwendig Werte per Hand in diese Spalte eingetragen hat und diesen Stand wieder herstellen möchte, kann er den globalen Befehl rückgängig machen.

3.4.2.4 WindloadsChart

Das `UserControl WindloadsChart` zeigt das Verhältnis von Höhe zu Windlast von Stäben gleichen Querschnitts an. In der Legende für die einzelnen Datenreihen wird also die Bezeichnung für die Querschnittseinstellung angezeigt (*REQ-022b*).

Zur Anzeige nutzt das `WindloadsChart` das online frei verfügbare Projekt Dynamic Data Display (CodePlex Open Source Community). Es sollte im weiteren Verlauf der Implementierung möglicherweise durch ein geeigneteres ersetzt werden, da die freie Bibliothek Schwächen in der gemeinsamen Darstellung von Linien- und Punktgraphen aufweist.

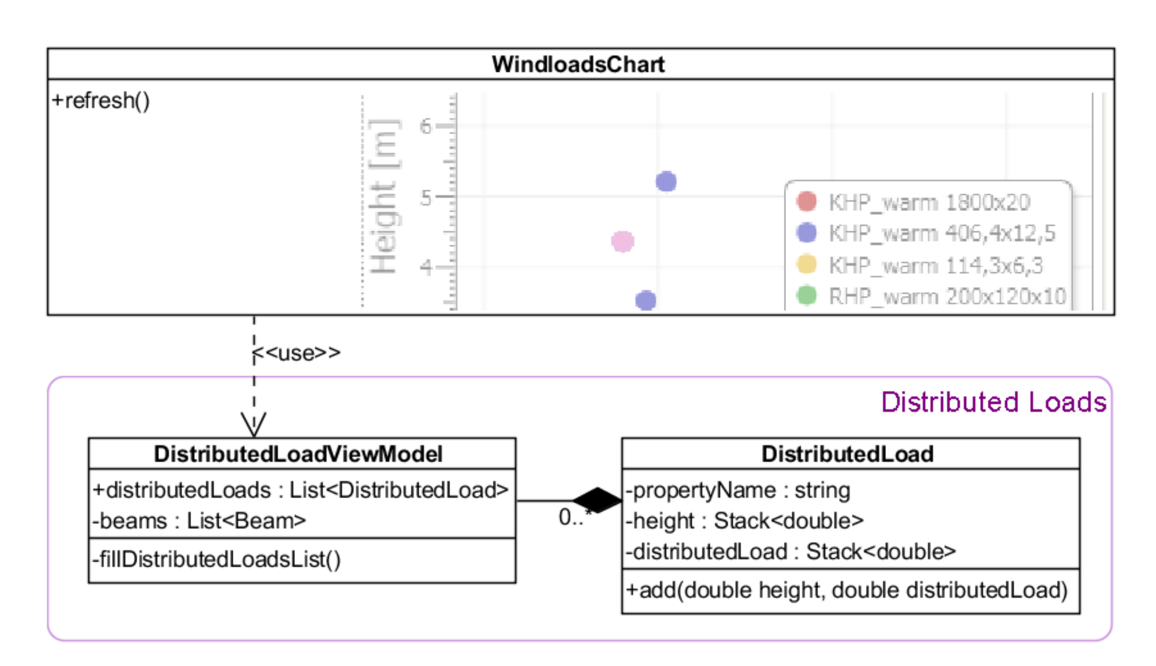


Abbildung 18: Klassendiagramm hinter dem UserControl WindloadsChart

In Abbildung 18 wird die Klassenstruktur hinter dem `WindloadsChart` gezeigt. Das `UserControl` greift auf das ViewModel für Streckenlasten (engl. „distributed loads“) zu. Das `DistributedLoadViewModel` bekommt bei der Initialisierung die Referenz auf die Liste der Stäbe übergeben, die in `WindloadsTable` angezeigt werden. Aus dieser Liste wird aus den Einzellasten an den Stabenden eine Streckenlast für jeden Stab berechnet, die in einer Liste aus `DistributedLoad`-Objekten gespeichert wird. So ein Objekt enthält die drei Felder `propertyName` (Querschnittsbezeichnung), `height` (mittlere Stabhöhe) und `distributedLoad` (Streckenlast, also Last pro Längeneinheit). Im `WindloadsChart` ergeben `height` und `distributedLoad` die Koordinaten eines Punktes,

während Punkte mit gleichem Wert für `propertyName` in einer Datenreihe zusammengefasst werden.

3.4.2.5 ApplyLoadsToModel

Im unteren Bereich des `Windloads`-Tabs lassen sich die berechneten Lasten zusammenfassen und als Lastfall in der Straus-Modelldatei abspeichern (*REQ-022c* und *REQ-020*). Aktuell ist das Hinzufügen eines Lastfalls mit neuem Namen möglich, langfristig soll es an dieser Stelle auch die Möglichkeit geben, Lasten zu einem bestehenden Lastfall hinzuzufügen.

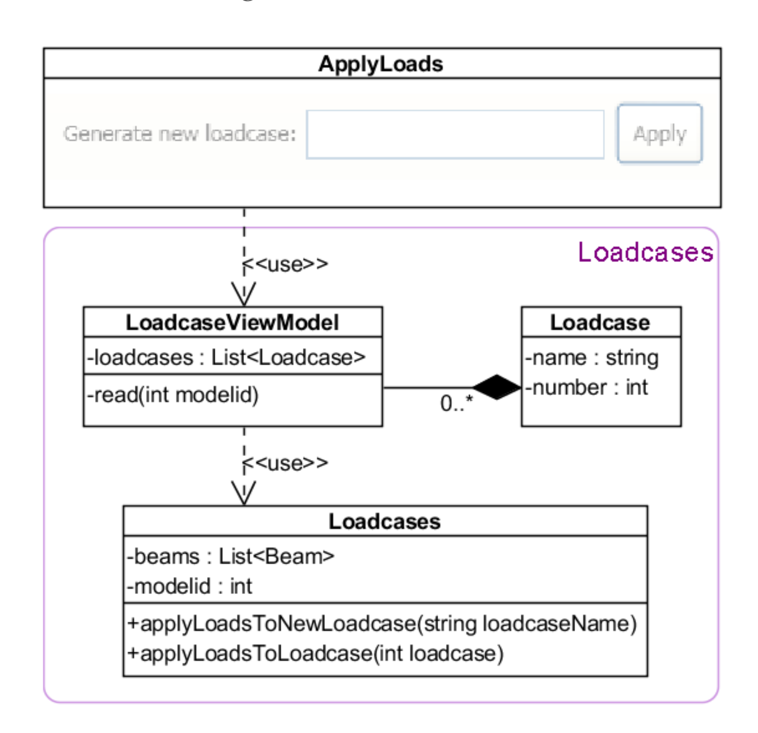


Abbildung 19: ApplyLoads-UserControl und beteiligte Klassen

In Abbildung 19 ist die Klassenstruktur hinter dem `UserControl ApplyLoadsToModel` dargestellt. Sie ist schon so ausgelegt, dass bestehende Lastfälle aus dem Modell ausgelesen und angezeigt werden können. Aktuell greift die Oberfläche auf die Methode `applyLoadsToNewLoadcase()` zu, um einen neuen Lastfall zu erstellen. Die bestehenden Lasten sollen später über ihren Namen mit Hilfe einer `ComboBox` identifiziert werden, damit auch ihnen berechnete Lasten zugeordnet werden können.

3.4.3 Testbarkeit

Das Testen der Oberfläche soll mit Hilfe von Integrationstests durchgeführt werden. Diese können zunächst manuell durchgeführt und später automatisiert werden. Dieser Abschnitt soll sich vor allem mit der Testbarkeit des Daten verarbeitenden Codes beschäftigen.

Wie schon im Abschnitt zur `Straus7`-Komponente beschrieben, erweist sich das Testen von Methoden, die auf die Straus-API zugreifen, als umständlich. Die gewählte Vorgehensweise soll im Folgenden anhand des Unit-Tests für die `Windloads`-Klasse exemplarisch gezeigt werden, da diese für die sicherheitsrelevanten Berechnungen zuständig ist. In Abbildung 20 ist die Klassenstruktur für den Test dargestellt.

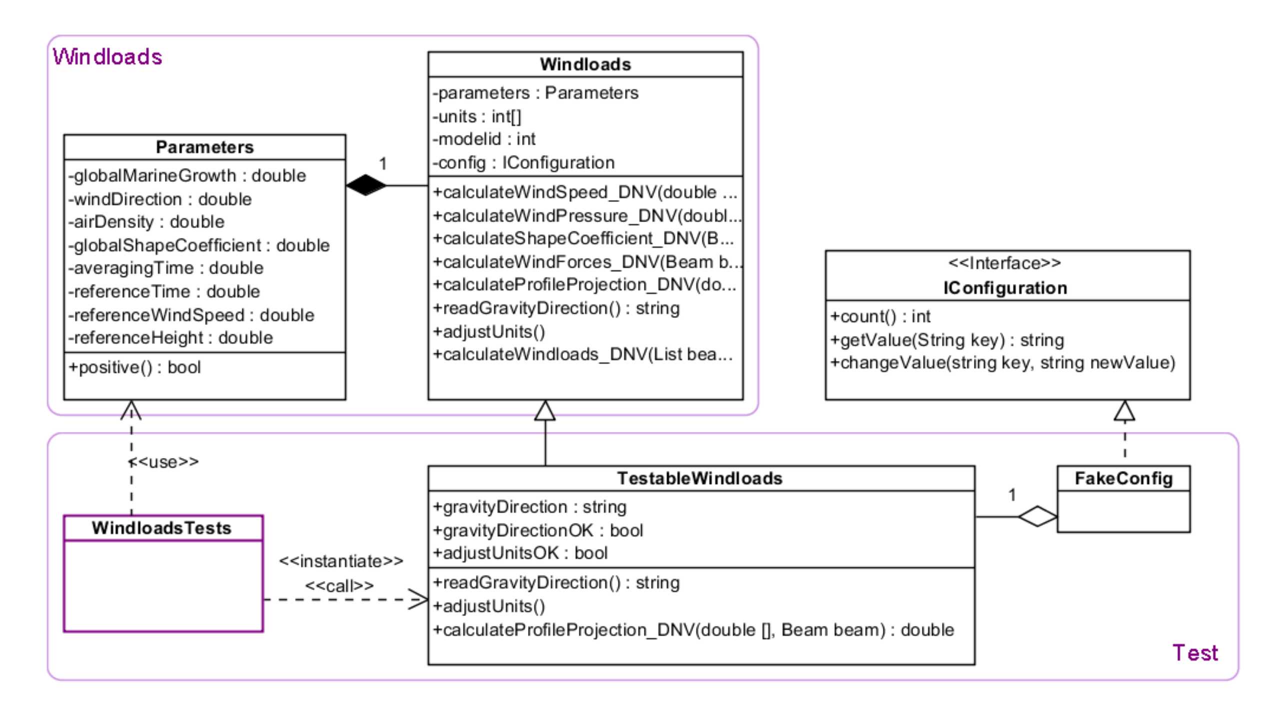


Abbildung 20: Testbarkeit der Windloads-Klasse mit Unit-Tests

Wie in der Abbildung zu sehen, wird im Testprojekt die `TestableWindloads`-Klasse definiert, die von der `Windloads`-Klasse erbt. Diese Klasse übernimmt alle Eigenschaften der Vaterklasse, überschreibt aber die drei Methoden, die auf die Straus-API zugreifen mit Fake-Methoden. Da die `Windloads`-Klasse eine Referenz auf ein Objekt benötigt, das `IConfiguration` implementiert, wird im Testprojekt ein Fake-Objekt der Klasse `FakeConfig` erstellt, das nur diesen Zweck erfüllt. Die originale `Windloads`-Klasse nutzt die `Configuration`-Komponente um die Richtung der Schwerkraft auszulesen, die in der Modelldatei eingestellt ist. In der `TestableWindloads`-Klasse wird die Richtung im Feld `gravityDirection` abgelegt und ist an dieser Stelle auch für den Test konfigurierbar. Die Testfälle der Klasse `WindloadsTest` testen die abgeleitete Klasse `TestableWindloads` anstelle der Original-`Windloads`-Klasse. Dazu nutzt sie außerdem die Klasse `Parameters`, mit deren Hilfe sie beliebig konfigurierte `Parameters`-Objekte erstellen kann, die sie der `TestableWindloads`-Klasse zur Berechnung mitgeben kann.

Auf diese Weise können die Berechnungen der `Windloads`-Klasse unabhängig von den Zugriffen auf die Straus-API getestet werden. Die Testbarkeit ist an dieser Stelle ausgesprochen wichtig, weil hier die sicherheitskritischen Berechnungen durchgeführt werden, mit deren Hilfe die Straus-Modelle auf Stabilität geprüft werden (*REQ-039*).

Was die anderen Klassen der Komponente angeht, ist die Klasse `DistributedLoads-ViewModel`, die aus der Liste der vorhandenen Stäbe die Streckenlasten berechnet, auf ähnliche Weise testbar wie die `Windloads`-Klasse. Auch für sie wurde eine Testklasse erstellt, die einen beispielhaften Testfall enthält.

Auf herkömmlichen Weg nicht sinnvoll mit Unit-Tests testbar sind die `UserControls`, sowie die Klassen `LoadCases` und `LoadCaseViewModel`, weil sie in jeder Methode externe Zugriffe beinhalten. Hier kann auch nur auf Integrationstests oder die Verwendung des Moles-Frameworks zurückgegriffen werden. Ähnlich sieht es mit der Klasse `BeamReader` aus, hier arbeitet lediglich die Methode `beamProfileType-ToString()` ohne externe Zugriffe.

Weiterhin gibt es eine Reihe von Klassen, bei denen Unit-Tests nicht zwingend nötig sind, weil sie keine Logik beinhalten, sondern nur Aufrufe. Dabei handelt es sich um die Klassen `Beam`, `BeamDataViewModel`, `DistributedLoad`, `WindloadsInstruction-Factory`, `RecalculateWindloads`, `SetGlobalMarineGrowth`, `Loadcase`, `Parameters` und `Wind-loadCommands`. Diese Klassen werden sinnvollerweise im Rahmen von Integrationstests geprüft.

3.4.4 Designentscheidungen

Auch für die `Windloads-Mission` und die `Mission`-Komponente im Allgemeinen wurden wichtige Entscheidungen getroffen, die hier begründet werden. Alle Entscheidungen sollten bei der Implementierung weiter `Missions` bedacht werden.

Verwenden von `UserControls` zur Einbindung von `Mission`-Komponenten. Die Einbindung der `Mission`-Komponenten rein auf der Ebene der Benutzeroberfläche erlaubt eine größtmögliche Unabhängigkeit der Funktionalität vom Hauptprogramm, deshalb wurde diese Schicht als Schnittstelle genutzt. Eine weitere Möglichkeit wäre das Einbinden über ein sogenanntes `ResourceDictionary` auf XAML-Ebene. Ein `ResourceDictionary` kann beliebige XAML-Elemente beinhalten und in eine andere Datei auslagern. Auch Code-Behind-Dateien sind möglich. Allerdings erscheint der Weg über `UserControls` sauberer und übersichtlicher, weil sie direkt die passende Struktur für diesen Anwendungsfall anbieten.

Verwenden einer Fabrik-Klasse. Das Generieren von Instructions in dieser Komponente entspricht keinem Erzeugermuster von (Gamma, et al., 2004). In Anlehnung an den Musterkatalog werden jedoch in der `InstructionsFactory`-Klasse Methoden genutzt, die unterschiedliche Instruction-Objekte erzeugen. Dies dient dazu, die Menge an Objekt-Referenzen, die von den Instructions benötigt werden, zu reduzieren. Durch Einführung der Fabrik müssen die aufrufenden Klassen diese Referenzen nicht halten, sondern kennen nur das Fabrik-Objekt, das wiederum die passenden Referenzen hält.

Auslagern von API-Aufrufen in eigene Methoden. Diese Entscheidung wurde ausführlich im Abschnitt „Testbarkeit“ erläutert. Der Grund für die Auslagerung ist, dass die ausgelagerten Methoden, wenn sie als `virtual` gekennzeichnet werden, von einer erbenden Klasse überschrieben werden können. Damit werden die anderen Methoden dieser Klasse für Unit-Tests isolierbar und testbar.

3.5 ModelManagement

Diese Komponente wird nur einmal instanziiert, weil nur ein Modell zurzeit geöffnet sein kann. Sie ist für das Öffnen, Speichern und Schließen der Modelldatei zuständig (*REQ-026* und *REQ-050*). Dazu werden die Funktionen der Straus-API genutzt (*REQ-023*). Außerdem werden beim Öffnen einer Datei die enthaltenen Einheiten gesichert und mit dem Speichern der Datei wieder hergestellt, damit die Einheiten-Einstellungen in der Modelldatei für den Benutzer unverändert bleiben.

Die möglichen Fehler wie „Datei bereits geöffnet“, „Keine Datei mit diesem Namen“ etc. werden als `Exception` verpackt und an den Aufrufer weitergegeben.

3.5.1 Vor- und Nachbedingungen

Um diese Komponente nutzen zu können, muss eine gültige Straus-API-Lizenz vorhanden und registriert sein. Das Straus-Programm muss installiert sein. Nach Verwendung der Model-Management-Komponente kann – je nach Reihenfolge und Art der benutzten Methoden – das Programm in einem beliebigen Zustand sein. Siehe dazu auch das Zustandsdiagramm in Abschnitt 3.5.4.

3.5.2 Struktur

Die `ModelManagement`-Komponente besteht nur aus einer Klasse und zwei Interfaces (siehe Abbildung 21). Die Unterscheidung der beiden Interfaces dient dem Zweck, dass andere Komponenten mit unterschiedlichen fachlichen Anforderungen auf die Klasse `ModelManager` zugreifen. Die `Ambience`-Komponente nutzt das Interface

`IManageModel`, sie ist die einzige, die das Öffnen, Speichern und Schließen von Modell-Dateien veranlasst. Die `Mission`-Komponenten nutzten das Interface `IModel`, über das sie die ID der geöffneten Modelldatei erfragen können, die sie für die Kommunikation mit der Straus-API benötigen. Darüber hinaus können sie abfragen, ob die API bereits initialisiert wurde und können das Feld `modified` auf `true` setzen, sobald sie das Modell im Arbeitsspeicher verändert haben (vgl. Abschnitt 2.3.5).

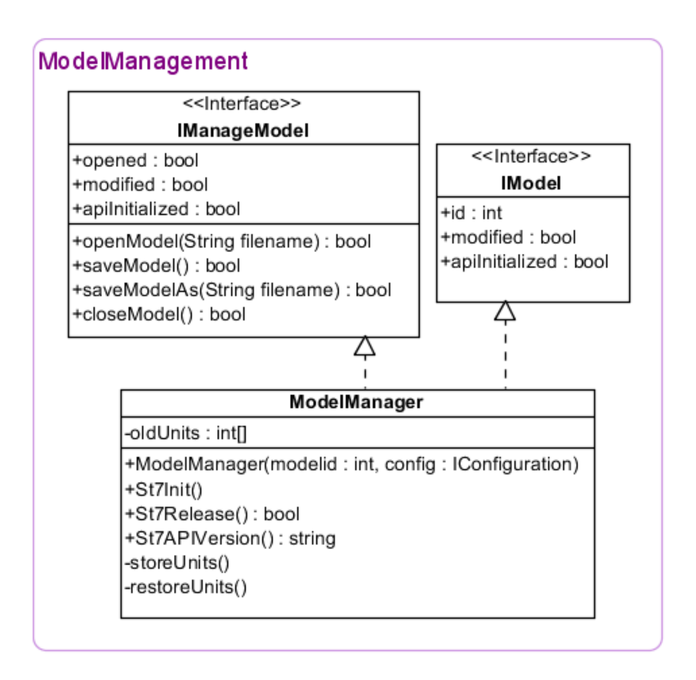


Abbildung 21: Klassenstruktur der ModelManagement-Komponente

Die `Ambience`-Komponente kann auf das Feld `modified` über `IManageModel` nur lesend zugreifen. Sie entscheidet anhand des Wertes von `modified`, ob sie den Speichern-Button aktiviert und ob beim Schließen des Programms nachgefragt wird, ob das Modell gespeichert werden soll.

3.5.3 Testbarkeit

Die Komponente `ModelManagement` ist nur mit Hilfe von Integrationstests oder einem Framework wie Moles testbar. Das liegt daran, dass jeder Teil dieser Komponenten auf die Straus-API zugreift.

3.5.4 Designentscheidungen

Nutzung von Zustandsvariablen. Das Verhalten von `Ambience` und `Mission` hängt in einigen Situationen davon ab, welchen Zustand das Modell und die API-Schnittstelle haben. Deshalb wurden diese Zustände mit Hilfe von booleschen Variablen modelliert. Die Zustände und ihre Transitionen sind in Abbildung 22 dargestellt.

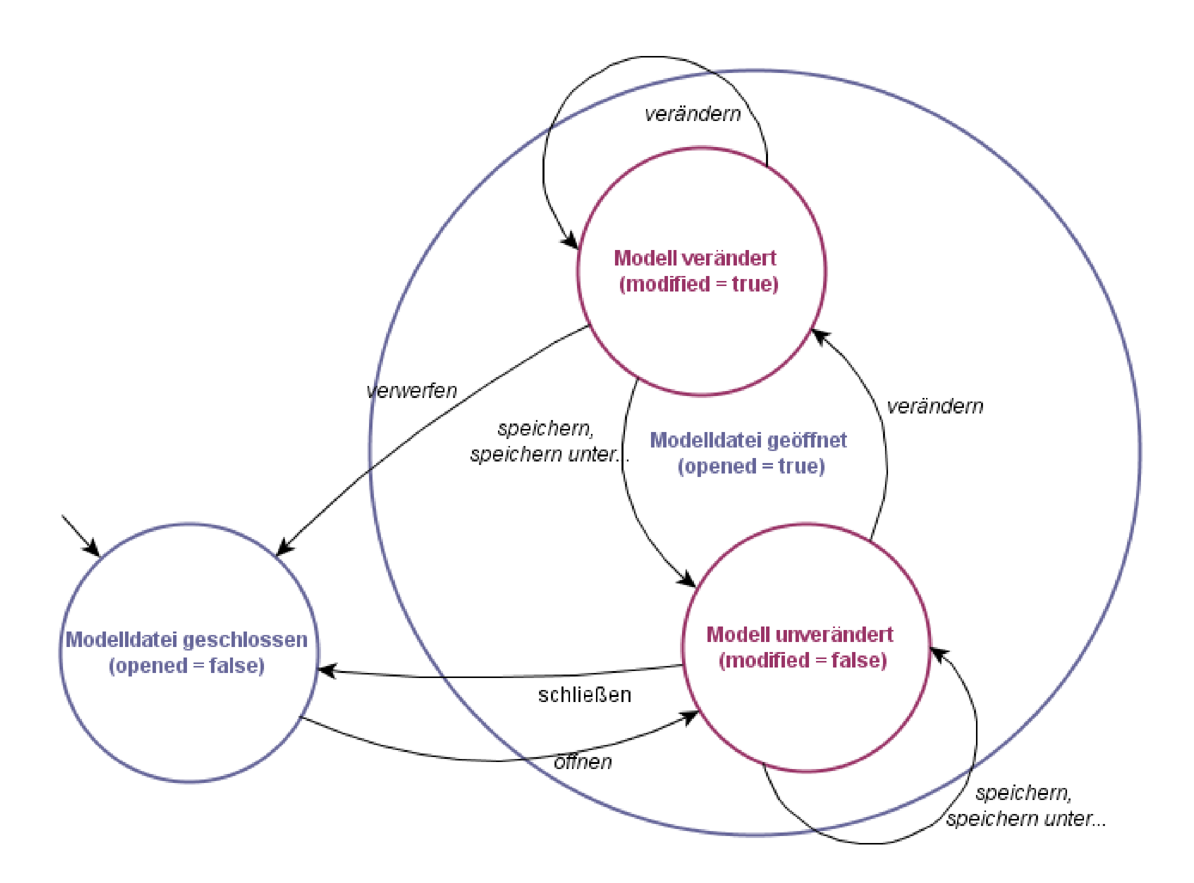


Abbildung 22: Zustände innerhalb der Klasse ModelManager

Keine vollständige Entsprechung des Straus-Modells in Software. Würde das Straus-Modell vollständig in Software modelliert werden, wie man *REQ-019* auch interpretieren könnte, wäre die `ModelManagement`-Komponente der richtige Ort dafür. Allerdings wird ein geöffnetes Straus-Modell nach Angaben des Strand7-Supports bereits durch die API im Arbeitsspeicher gehalten. Ein vollständiges Abbild zu reproduzieren, würde also unnötig viel Arbeitsspeicher kosten und die Wartezeiten für den Benutzer verlängern. Deshalb wurde entschieden, nur die Teile des Strukturmodells im Straus Multitool zu modellieren, mit denen aktuell gearbeitet werden soll. Dies geschieht dann in der jeweils passenden Form in den `Mission`-Modulen.

In den Anforderungen wird das **Verarbeiten von Ergebnisdateien** gefordert (*REQ-021*). Eine Ergebnisdatei beinhaltet das Ergebnis einer Simulation innerhalb der Straus-Software. Wenn die Umsetzung von *REQ-021* geplant wird, kann `ModelManagement` als Blaupause für eine `ResultManagement`-Komponente dienen. Diese würde die Funktionen öffnen, speichern und schließen für Ergebnisdateien zur Verfügung stellen.

Um die **Datensicherheit** zu verbessern (*REQ-52*) besteht die Möglichkeit, beim Öffnen der Datei die Komponente zunächst einmal mit einer lokalen Kopie arbei-

ten zu lassen. Damit könnte garantiert werden, dass das Straus Multitool auch bei Fehlerzuständen keine Modelldatei beschädigt. Die zusätzliche Sicherung wurde bisher von Overdick als unnötig eingestuft, da die Modelldatei erst mit dem Speichern-Befehl wirklich verändert wird. Änderungen, die nicht gespeichert wurden, werden bei Programmende verworfen.

3.6 Ambience-Komponente

Die `Ambience`-Komponente ist enthält die zu Grunde liegende Oberfläche für das Programm. Sie beinhaltet die grundlegenden Steuerelemente und ruft weitere Programmteile auf. Außerdem ist sie für Benutzerdialoge im Fehlerfall zuständig.

Wenn das Straus Multitool auch mit Straus-Ergebnisdateien arbeiten können soll (vgl. *REQ-021* und Abschnitt 3.5.4), soll hierfür soll keine neue Oberflächenstruktur geschaffen werden. Die Dateien sollen über die vorhandenen Buttons geöffnet und geschlossen werden können. Die hinzukommende Ablauflogik wird über den Kontext gelöst, in dem die Befehle verwendet werden.

3.6.1 Vor- und Nachbedingungen

Die `Ambience`-Komponente wird von der `Initialization`-Komponente als letztes initialisiert und erwartet Referenzen auf drei Objekte, von denen jeweils eins `IHistory`, `IConfiguration` und `IModelManager` realisiert. Wird das Programm geschlossen, beendet die `Ambience`-Komponente die Verbindung zur Straus-API, nachdem sie eine eventuell geöffnete Straus-Datei geschlossen hat.

3.6.2 Struktur

Das Klassendiagramm der `Ambience`-Komponente ist in Abbildung 23 dargestellt. Durch die Entscheidung WPF für die Beschreibung der Oberfläche einzusetzen, entspricht das Klassendiagramm nicht dem UML-Standard. Im Klassendiagramm wurde für das `ConfigurationWindow` keine getrennte Darstellung gewählt, da sie im Durchstich nicht ausprogrammiert wurde. Trotzdem besteht sie aus einer XAML-Datei für die Fensterbeschreibung und einer Code-Behind-Datei.

Die Klasse `MainWindow` ist die Code-Behind-Datei für die in Abbildung 23 als Klasse dargestellte `MainWindow.xaml`. Die Klassendarstellung für die XAML-Datei wurde gewählt um die Abhängigkeiten klar ausdrücken zu können.

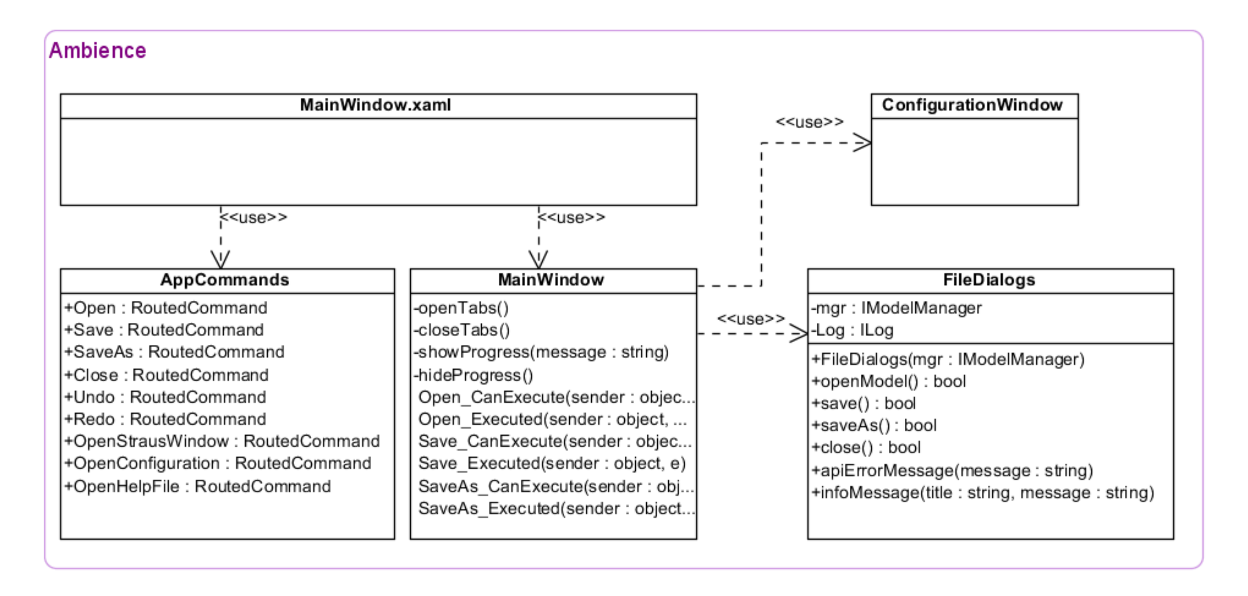


Abbildung 23: Klassendiagramm der Ambience-Komponente

Um die Struktur verständlicher zu machen, wird hier eine kurze Einführung in die Funktionsweise von selbst geschriebenen `RoutedCommands` in WPF gegeben, die von der Oberfläche verwendet werden: Zunächst wird in der Klasse `AppCommands` ein neues Objekt der Klasse `RoutedEvents` instanziiert. In der Klasse `MainWindow` gibt es dazu zwei Methoden, die sich um die Ausführbarkeit (`CanExecute`) und die Ausführung (`Executed`) kümmern. Die Bindung des Kommandos an Schaltflächen und an die ausführenden Methoden übernimmt die XAML-Datei, die die Oberfläche beschreibt (`MainWindlow.xaml`).

Die Initialisierung der `Mission`-Module erfolgt durch die Methode `openTabs()` in der `MainWindow`-Klasse. Hier wird für jedes `Mission`-Modul überprüft, ob dessen Vorbedingungen erfüllt wurden. Ist dies der Fall, wird das Modul in die Darstellung des Programms eingebunden.

3.6.3 Testbarkeit

Da es sich bei der `Ambience`-Komponente um eine reine Realisierung der Oberfläche und deren Funktionen handelt, ist sie nicht durch Komponententests überprüfbar. Deshalb wurde die Testbarkeit beim Design nicht weiter berücksichtigt. Zwei Möglichkeiten, die Komponente dennoch zu testen sind manuelle Tests und automatisierte Funktionaltest. Die Testdurchführung liegt allerdings außerhalb des Rahmens dieser Arbeit.

3.6.4 Designentscheidungen

Verwendung des MVVM-Musters und WPF (*REQ-002*). Die WPF-Version des Model-View-Controller-Musters ist das MVVM-Muster. Die Abkürzung steht für

Model-View-ViewModel. Das Muster schreibt vor, dass es ein Model gibt, das die Fachlogik implementiert. Dieses entspricht dem Model beim Model-View-Controller-Muster. Außerdem gibt es ein oder mehrere ViewModels, die die darzustellenden Sichten auf das Model modellieren. Dieses ViewModel kann zum Beispiel eine Liste von Objekten sein, die dann vom View als `ComboBox`, Tabelle oder Graph dargestellt werden. Da WPF für das .NET-Framework der aktuelle Stand der Technik ist und durch dessen Anwendung automatisch eine leichte Austauschbarkeit der Benutzeroberfläche gewährleistet wird, wurde diese Technik ausgewählt. Für weitere Informationen zu WPF und dem MVVM-Muster siehe (Huber, 2010).

Verwendung eines Ribbon-Menüs verworfen. Im Designprozess kam der Gedanke auf, die Oberfläche nicht in Tabs aufzuteilen, sondern für die verschiedenen Steuermöglichkeiten ein Ribbon-Menü einzufügen, wie es Microsoft für aktuelle Office-Versionen entwickelt hat. Dagegen spricht, dass das Konzept des Ribbons ist, *ein* bearbeitetes Objekt z. B. ein Dokument zu haben, das konstant im Hauptbereich der Oberfläche angezeigt wird. Die Methoden zum Arbeiten mit dem Objekt sind in dem Ribbon nach unterschiedlichem Kontext angeordnet. Das Straus-Multitool benötigt für die unterschiedlichen Module auch unterschiedliche Darstellungen des Modells. Damit würde sich beim Wechseln der Ansicht zu einem anderen Modul nicht nur das Menü, sondern auch die Ansicht des Objekts ändern müssen. Dafür ist das Ribbon aber nicht konzipiert. Deshalb wurde die Idee verworfen.

Nutzung von `RoutedCommands` für Befehle. `RoutedCommands` bieten einen Mehrwert zu einfachen Events. Neben einigen anderen Vorteilen, die für das Straus Multitool nicht entscheidend sind, bieten `RoutedCommands` die Möglichkeit, die Ausführbarkeit der Kommandos zentral zu regeln. Zum Beispiel müssen ein Button für `Undo` und ein Programmstück, dass auf die Tastenkombination Strg+Z reagiert, nicht einzeln aktiviert und deaktiviert werden, wenn nichts mehr rückgängig gemacht werden kann. Die Logik hierfür liegt zentral an einer Stelle und gilt für das `RoutedCommand`. Durch die Bindung des `RoutedCommands` an Schaltflächen und Tastenkombinationen, geschieht die Aktivierung und Deaktivierung der Funktonalität automatisch. Das erhöht die Wiederverwendbarkeit des Codes. Thomas Claudius Huber beschreibt den Grund für die Nutzung von `RoutedCommands` mit den folgenden Worten: „Natürlich ließe sich für jede Aktion auch ein Event Handler für ein bestimmtes Event installieren. Doch dann müssten Sie vieles ‚zu Fuß' erledigen, wie eben beispielsweise das Aktivieren und Deaktivieren von MenuItems oder Buttons. […] Dies

führt zu umfangreicherem Code, den Sie sich mit Commands bei der WPF teilweise oder in manchen Fällen auch ganz sparen können.“ (Huber, 2010 S. 469)

Mit der Verwendung von `RoutedCommands` wird auch die Austauschbarkeit der Programmoberfläche erleichtert. Denn das dahinter liegende Konzept hilft dabei, Programmlogik und Oberflächenlogik sauber zu trennen. Damit spricht auch dieser Punkt für die Nutzung von `RoutedCommands`, vgl. (Krypczyk, 2011).

3.7 Initialization-Komponente

Diese Komponente ist der Einstiegs- und der Endpunkt des Programms. Von hier aus werden alle anderen Vorgänge angestoßen. Beim Startvorgang wird ein Startbildschirm angezeigt, der dem Nutzer mitteilt, was im Hintergrund passiert. Die Komponente verrichtet dabei eine Reihe von Initialisierungsschritten und Überprüfungen. Da die Komponente nur diesen einen linearen Ablauf enthält, wird auf Struktur und Testbarkeit der Komponente nicht weiter eingegangen.

Eine Aufgabe der Komponente ist zu überprüfen, ob alle zur Ausführung notwendigen Programme und Bibliotheken zur Verfügung stehen. Im Fehlerfall gibt sie eine entsprechende Meldung aus. Dabei wird konkret das Vorhandensein der Straus7-API (Installation, gültiger Straus-Lizenzschlüssel) überprüft.

Die Komponente initialisiert außerdem die Komponenten `ModelManagement`, `Configuration` und `History` sowie die Oberflächenkomponente `Ambience`. Abbildung 24 zeigt den Ablauf der `Initialization`-Komponente in einem Aktivitätsdiagramm. Mit roten Pfeilen ist der Kontrollfluss dargestellt, die schwarzen Pfeile zeigen die Übergabe von Referenzen auf bereits initialisierte Komponenten an (Objektfluss).

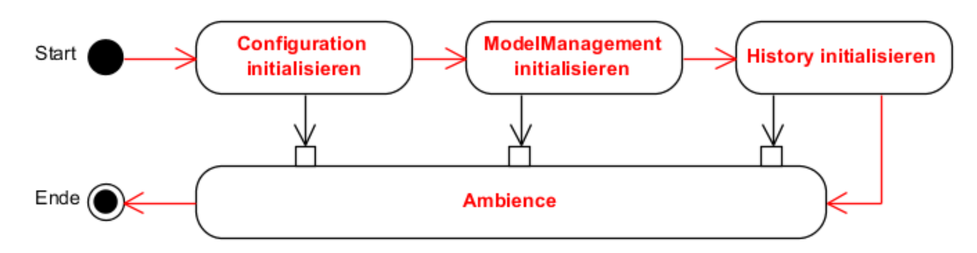


Abbildung 24: Die wichtigsten Schritte der Initialization-Komponente

Die `Initialization`-Komponente ist der Punkt, an dem langfristig die Überprüfung eines Lizenzschlüssels für das Straus Multitool stattfinden könnte. Außerdem ist sie die letzte Instanz für ungefangene `Exceptions`. Hier werden `Exceptions` spätestens gefangen und eine entsprechende Fehlermeldung für den Anwender generiert.

4 Technische Konzepte

In diesem Kapitel werden Bausteinübergreifende Themen vorgestellt, die wichtig für das Straus Multitool sind. Die Themen Persistenz, Ergonomie und Testbarkeit zählen auch zu den bausteinübergreifenden Themen, wurden aber bereits in Kapitel 2.1 beschrieben und werden daher hier nicht erneut betrachtet. Beschrieben wird hier der Umgang mit den Themen Fehlerbehandlung, Logging und physikalische Einheiten.

4.1 Ausnahme- und Fehlerbehandlung

In diesem Kontext sollen Fehler als Zustände betrachtet werden, die problematisch sind, mit denen das Programm aber weiter arbeitsfähig ist. Eine falsche Benutzereingabe ist ein passendes Beispiel. Mit dem Begriff der Ausnahme soll ein abnormer Zustand beschrieben werden, der dann entsteht, wenn das Programm in eine ausweglose Situation geraten ist. In dieser Situation kann das Programm nicht weiter arbeiten (Siedersleben, 2004).

Johannes Siedersleben bezeichnet den „Umgang mit Fehlern und Ausnahmen [als] eines der wichtigsten Themen der Software-Architektur“ (Siedersleben, 2004). Deshalb sollte prinzipiell gut überlegt sein, wie mit diesem Thema innerhalb der Architektur umgegangen werden wird. Siedersleben stellt ein Muster-Vorschlag unter Verwendung einer Sicherheitsfassade und einer Diagnose- und Reparatur-Schnittstelle für jede Komponente vor. Allerdings schränkt er ein: „Nicht jede Komposition braucht eine Sicherheitsfassade. In kleinen und mittleren Systemen wird eine einzige Sicherheitsfassade für den gesamten Anwendungskern ausreichen.“ (Siedersleben, 2004)

Für das hier entwickelte Straus Multitool wurde folgende Vorgehensweise gewählt: Auf den unteren Ebenen werden nur Fehler aufgefangen und behandelt. Diese Fehler werden in der Software als Exception abgebildet. In Form einer neuen Exception mit einer Beschreibung der Umstände und dem Ort des Auftretens wird der Fehler weiter gereicht. Fehler sind Probleme, die der Benutzer beheben kann, wie das Fehlen einer gültigen Straus-Lizenz, fehlerhafte Modelldatei etc. Diese Fehler werden dann auf der Präsentationsschicht dem Benutzer mitgeteilt. Die lokale Bearbeitung von Fehlern lässt sich dadurch begründen, dass ihre angemessene Behandlung dem Entwickler im Kontext eines Methodenaufrufs deutlich klarer ist, als wenn die Behandlung zentralisiert wäre.

Benutzereingaben werden in den Code-Behind-Dateien zu den Eingabeelementen geprüft und validiert. Trotzdem sollen Methoden Werte zurückweisen, die sie nicht verarbeiten können und in einer passenden `Exception` den Fehler beschreiben.

Ausnahmen werden bis zur Sicherheitsfassade „hinaufgereicht". Diese kümmert sich um einen entsprechenden Dialog mit dem Benutzer und um ein ordnungsgemäßes Beenden des Programms.

4.2 Logging

Als Technik für die Protokollierung wurde die Entscheidung zugunsten der Log4NET-Bibliothek von Apache getroffen. Diese Bibliothek ist online unter einer freien Lizenz verfügbar, vgl. (Apache Software Foundation).

Normalerweise würde man versuchen, das Logging über eine eigene Komponente zu kapseln, damit die Bibliothek leicht austauschbar ist und die direkten Aufrufe sich auf einen lokalen Bereich beschränken. Dagegen spricht jedoch, dass die Log4NET-Bibliothek protokollieren kann, welche Methode den Logging-Befehl aufgerufen hat. Diese hilfreiche Mehrinformation müsste bei einer Kapselung über Reflection weiter gegeben werden. Insgesamt wurde eine Kapselung der Logging-Komponente wegen einem zu hohen Aufwand verworfen.

Ein großer Vorteil beim Benutzen der externen Bibliothek ist der große Funktionsumfang und die gute Konfigurierbarkeit des Systems. Außerdem ist die Fremdsoftware weiter entwickelt und somit weniger fehleranfällig als eine Neuentwicklung.

4.3 Einheiten

In der Oberfläche werden die physikalischen Einheiten zu jeder angezeigten Größe (Ein- und Ausgabe) vermerkt. Im Quellcode werden die Einheiten nicht einzeln modelliert. Es wird direkt mit Zahlenwerten gerechnet, Einheiten sollten möglichst ständig in den Kommentaren mitgeführt werden. Dieser Umgang mit physikalischen Einheiten ist für gewöhnlich stark fehleranfällig, deshalb soll im Folgenden beschrieben werden, warum er dennoch gewählt wurde.

Zunächst gibt es unterschiedliche Wege für den Umgang mit physikalischen Einheiten. Der objektorientierte Ansatz wäre, die Größen als Objekte zu modellieren. Damit entsteht ein Entwicklungsaufwand für die Skalierung der Größen. Zum Beispiel bei der Umrechnung von Pascal in Mega-Pascal. Für jede Größe müssten

alle diese Umrechnungen implementiert werden. Viel schwerer wiegt aber der Umgang mit zusammengesetzten Einheiten. Die Größen müssten Methoden implementieren, die ihre mathematische Kombination ermöglichen. Dabei müssen ein Zahlenwert und eine sinnvolle Einheit herauskommen. Bei diesen Vorgängen gibt es sehr viele mögliche Kombinationen, die die Implementierung dieses Grundgerüstes sehr fehleranfällig machen.

Einige existierende Bibliotheken, die dieses Problem adressieren, wurden oberflächlich untersucht. Der Evaluierungsaufwand für eine sichere Entscheidung hat sich dabei als zu groß erwiesen.

In der Programmiersprache F# werden solche Rechnungen nativ unterstützt. Obwohl diese Sprache direkt in das Programm integrierbare wäre, wurde auf die Einführung einer weiteren Programmiersprache verzichtet um die Komplexität nicht unnötig zu vergrößern und die Wartbarkeit nicht einzuschränken.

Die Situation für die Handhabung der physikalischen Größen als einfache Zahlenwerte wird dadurch entschärft, dass eine Umwandlung der Einheiten über die Straus-API jederzeit möglich ist. Die Lösung sieht nun so aus, dass die im Modell eingestellten Einheiten beim Öffnen einmal zwischengespeichert und beim Schließen wiederhergestellt werden. Damit merkt der benutzende Ingenieur nichts von der internen Umstellung. Damit kann aber an jeder Stelle im Programm, an der eine Berechnung erfolgen soll, zunächst bequem die passende Umgebung hergestellt werden: Als erstes werden die Einheiten im Modell eingestellt, die für die Rechnung passend erscheinen. Dann werden die benötigten Größen aus dem Modell gelesen, direkt mit den passenden Einheiten. Die Rechnung wird durchgeführt und das Ergebnis kann direkt gespeichert oder weiter verarbeitet werden. Die Einheit der Ergebnisgröße kann leicht ermittelt werden.

Deshalb soll beim Programmieren nicht davon ausgegangen werden, dass die physikalischen Einheiten auf eine bestimmte Weise eingestellt sind. Jede Rechnung stellt sich die passenden Einheiten im Modell über eine passende Funktion selbst ein, bevor die Berechnung erfolgt. Die passende Funktion verwendet den Straus-Aufruf `St7SetUnits()` und ist als `virtual` gekennzeichnet, damit sie für Komponententests überschrieben werden kann.

5 Fazit

Im Rahmen dieser Arbeit ist ein lauffähiges Programm entstanden, das die entwickelte Architektur illustriert. In diesem Kapitel wird der Status Quo betrachtet und bewertet sowie ein Ausblick auf die Weiterentwicklung der Anwendung gegeben.

5.1 Ergebnis

Das Straus Multitool erfüllt die für diese Arbeit geplanten funktionalen Anforderungen mit wenigen Einschränkungen. Es gibt eine Komponente `Configuration`, die für die Persistenz des Programms zuständig ist. Die Komponente `ModelManagement` übernimmt die Modellverwaltung. Die `History`-Komponente sorgt für eine zentrale Speicherung ausgeführter Befehle, die zurückgenommen und wiederholt werden können. Es wurde eine `Mission`-Komponente implementiert, die aus vorgegebenen Parametern Windlasten berechnet, grafisch darstellt und in der Modelldatei speichern kann. Die Programmschritte werden mit Hilfe der Log4NET-Bibliothek grob protokolliert.

Die Bewertung der nicht funktionalen Aspekte des Programms und der entwickelten Architektur erfordert eine etwas umfangreichere Analyse. Im Folgenden sollen die nicht-funktionalen Aspekte priorisiert und ihre Erfüllung bewertet werden. Da diese von Overdick eher weich formuliert wurden, kann diese Bewertung nicht rein objektiv erfolgen. Alle nicht-funktionalen Anforderungen wurden eingangs genannt: Änderbarkeit bzw. Erweiterbarkeit, Testbarkeit und Verständlichkeit. Diese Punkte sind auch in den in Kapitel 2.1 benannten Qualitätskriterien an eine Software-Architektur im Allgemeinen enthalten. Hinzu kommen als relevante, nicht-funktionale Aspekte noch Zuverlässigkeit, Benutzbarkeit und Effizienz. Diese sechs Punkte sollen im Folgenden zur Bewertung herangezogen werden. Dabei wird mit Hilfe der Skala *A = „wichtig“, B = „mittel“, C = „weniger wichtig”* priorisiert, die Begründung für die Einstufungen wird in Kapitel 2.1 gegeben. Die für diese Architektur nicht relevanten Punkte werden hier nicht aufgezählt.

5.1.1 Erweiterbarkeit (A)

Die Erweiterbarkeit der Architektur ist ein sehr wichtiges Kriterium. Das Straus Multitool soll problemlos durch neue Funktionen erweitert werden können, ohne dass sich die Struktur ändert. Die Erweiterbarkeit lässt sich schwer messen. Ein geeigneter Ansatz zur Bewertung dieses Aspekts scheint der Weg über sogenann-

te Qualitätsszenarien nach (Starke, 2011) zu sein. Diese ausführlich auszuarbeiten würde über den Rahmen dieser Arbeit hinausgehen. Durch die für später angesetzten Anforderungen sind allerdings schon einige wichtige geplante Erweiterungen bekannt. Diese sollen hier kurz betrachtet werden:

REQ-001 fordert eine modulare Bauform des Programms. Das Straus Multitool soll problemlos durch neue Module, die eine bestimmte Aufgabe erfüllen, erweitert werden können. Dies ist durch die Konzipierung der `Mission`-Komponenten umgesetzt worden. Dabei wurde keine Plugin-Architektur implementiert in dem Sinne, dass ein Hauptprogramm ergänzende Plugin-Komponenten automatisch einbinden würde. Die `Mission`-Komponenten werden direkt geladen, die Änderung zur Einbindung einer neuen `Mission` ist aber nur an einer Stelle des Quellcodes erforderlich. Dies wirkt sich positiv auf die Flexibilität aus, weil der Programmierer bei jeder neuen `Mission`-Komponente entscheiden kann, ob sie in einen Tab eingebunden werden, ein neues Fenster öffnen oder einfach im Hintergrund ausgeführt werden soll. Anforderungen, deren Umsetzung durch Einführung der `Mission`-Komponente einfach möglich werden, sind *REQ-045* „Wellenlasten generieren" und damit *REQ-024* „Waveloads.dll" und *REQ-037* „Extreme Lasten auswählen". Hinzu kommen die Anforderungen *REQ-029* „Joint Check", *REQ-028* „Beam-Buckling-Check", *REQ-030* „Auffinden von unverbundenen Stäben", *REQ-033* „Selektieren von Stäben bestimmter Länge", *REQ-035* „Clean Mesh", *REQ-036* „Tekla-Import" und *REQ-040* „Lastfall-Kombinationen generieren". Das Generieren der Windlasten nach DIN-Norm (*REQ-022a*) kann als zusätzliche Funktion der `Windloads-Mission` implementiert werden.

REQ-018 „Makros aufzeichnen" kann leicht mit Hilfe der `History`-Komponente realisiert werden. Die Anforderung *REQ-038* „Verschiedene Pfade pro Funktion" lässt sich einfach in der `Ambience`-Komponente und in der `Mission`-Komponente umsetzen. Durch die Nutzung von `RoutedCommands` (siehe Abschnitt 3.6.4) ist dies sogar besonders einfach. Die Umsetzung der Anforderung *REQ-007* „Konfigurationen speichern" ist mit der `Configuration`-Komponente möglich (siehe Abschnitt 3.1.4).

Die „Tabellenfunktionen" für das Kontextmenü einer Tabellenansicht (*REQ-053*) und die Möglichkeit „Daten als Funktionen an[zu]zeigen" (*REQ-043*) kann als Zusatzfunktion zum `UserControl WindloadsTable` hinzugefügt werden. Da diese Funktionen auch für andere Tabellendarstellungen in Frage kommen, wäre es hilfreich ein allgemeines `UserControl` für Tabellen einzuführen, das diese Funktionen zur Verfügung stellt und die speziellen Tabellenansichten von diesem erben zu lassen.

Auch die Anforderungen *REQ-054* „Werte in Tabelle visualisieren“ und *REQ-055* „Tabellenwerte filtern“ können auf diese Weise implementiert werden.

Falls das Arbeiten mit Ergebnisdaten weiterhin gewünscht wird (diese Entscheidung steht zum Erstellungszeitpunkt dieser Arbeit noch aus), sollten die zugehörigen Anforderungen in einer eigenen Komponente umgesetzt werden. Dabei kann für *REQ-021* „Ergebnisdaten verwalten“ die `ModelManagement`-Komponente zum Vorbild genommen werden. Die Anforderung *REQ-005* „Automatischer Bericht“ könnte auch innerhalb dieser Komponente umgesetzt werden. Das Speichern des Modells als vorverformte Struktur (*REQ-031*) scheint allerdings eher in eine `Mission`-Komponente zu passen, da auf die Ergebnis- und auf die `ModelManagement`-Komponente zugegriffen werden müsste.

Die automatische Überprüfung auf verfügbare Updates (*REQ-016*) findet ihren Platz in der `Initialization`-Komponente. Die Installation des Updates sollte mit einem Installer-Paket gelöst werden. Die Überprüfung einer Benutzerlizenz als Programmschutz (*REQ-013*) kann ebenfalls in der `Initialization`-Komponente implementiert werden.

Infrastruktur-Komponenten können relativ problemlos ausgetauscht werden, da die Abhängigkeiten zwischen den Komponenten mit Hilfe des „Dependency Inversion Principle“ reduziert werden konnte, vgl. auch den gleichnamigen Artikel von (Martin). Dabei werden allerdings nicht – wie dort beschrieben – abstrakte Klassen, sondern Interfaces verwendet, die eine lose Kopplung zwischen den Komponenten erreichen.

Eine Mehrsprachigkeit könnte durch eine Anpassung der `Configuration`-Komponente einfach eingeführt werden. Dazu könnten eindeutige Schlüssel als Platzhalter definiert werden, die in den nachfolgenden XML-Tags in unterschiedliche Sprachen übersetzt werden.

Zwei Änderungen, die nicht zwangsläufig leicht umsetzbar wären, sind das Austauschen der vorhandenen Logging-Bibliothek Log4NET und Änderungen in der Straus-API. Letzteres wird aber durch die Auslagerung der Straus-Aufrufe in eigene Methoden (siehe Kapitel 3.3) erleichtert, da nur diese Methoden an sich ändernde Straus-Funktionen angepasst werden müssten.

Anhand der vorangegangenen Betrachtungen lässt sich schließen, dass das Straus Multitool nach aktuellem Kenntnisstand sehr gut erweiterbar ist. Einschränkungen gibt es lediglich im Bereich der Austauschbarkeit der Log4NET- und der

Straus7-Komponente. Eine Änderung in einer dieser Komponenten durch ein Update der Bibliotheken würde nur dann einen größeren Anpassungsaufwand mit sich bringen, wenn die neue Version nicht abwärtskompatibel ist. Beide Szenarien werden aber nicht als wahrscheinlich eingestuft.

5.1.2 Testbarkeit (A)

Das System soll leicht zu testen sein. Dies ist vor allem in den Programmteilen wichtig, in denen es um die sicherheitsrelevanten Berechnungen geht. Aber auch Infrastruktur-Komponenten, auf die sich das System verlässt, sollten gut getestet sein. An allen diesen Stellen wurden exemplarisch Unit-Tests angelegt, die mit Einführung eines genauen Testplans erweitert werden sollten. Die Testbarkeit der anderen Klassen kann durch Mock-Frameworks und Integrationstests erreicht werden. Die Integrationstests sollen zunächst manuell durchgeführt werden. Mit wachsender Anzahl von Testdaten durch die späteren Benutzer soll auch diese Form von Test automatisiert durchgeführt werden können. Um sicher zu stellen, ob die Funktionen die Anforderungen erfüllen, kann mit den Hilfsmitteln der Akzeptanztest-getriebenen Entwicklung, wie beispielsweise dem Framework FitNesse (Martin, et al.) gearbeitet werden.

Es kann somit von einer umfassenden Testbarkeit des Straus Multitools ausgegangen werden. Die genaue Ausarbeitung und Umsetzung eines umfassenden Testplans muss nach Abschluss dieser Arbeit erfolgen.

5.1.3 Verständlichkeit (A)

Die Verständlichkeit der Struktur des Straus Multitools wird mit diesem Dokument als Dokumentation der Architektur sichergestellt. Wichtig ist hierbei, dass auch die Entscheidungen, die zu der vorliegenden Architektur geführt haben, dokumentiert sind.

Der Quellcode wurde nach Anregungen von (McConnell, 2004) zu Struktur und Kommentaren so dokumentiert, dass er auch für neue Entwickler schnell verständlich werden sollte. Die Qualität der Kommentare ist nicht quantitativ messbar, allerdings lassen sich zur Beurteilung der Verständlichkeit des Quellcodes an sich Metriken hinzuziehen, die diesen Aspekt vergleichbar machen sollen. Um einzuschätzen, ob der Quellcode des Straus Multitools verständlicher ist als der des Straus API Tools, wurden mit Hilfe des freien Programms SourceMonitor (Campwood Software LCC) vergleichende Messungen durchgeführt. Diese werden im Folgenden beschrieben.

Es sollen drei Metriken für die Verständlichkeit zu Rate gezogen werden: Die prozentuale Anzahl der Zeilen mit Code-Dokumentation, die Komplexität nach (McConnell, 1993) und die mittlere Einrücktiefe der Codezeilen.

Die Zeilen mit Code-Dokumentation sind alle Zeilen, in denen die Kommentare ein Format haben, das von Microsoft als Basis für eine automatisch generierbare Code-Dokumentation vorgesehen ist. Einfache Kommentare werden dabei nicht mitgewertet. Das macht die Messung etwas unscharf, weil auch gewöhnliche Kommentare die Verständlichkeit des Codes deutlich verbessern können. So werden aber auskommentierte Blöcke von Code, wie sie im Straus API Tool häufiger vorkommen, nicht mitgewertet. Als Ergebnis der Messung liegt das neu entwickelte Straus Multitool mit 12,2% Code-Dokumentation deutlich vor dem Straus API Tool mit 2,5% Code-Dokumentation.

Die Komplexität nach Steve McConnell kann als maximaler und durchschnittlicher Wert berechnet werden. Die Werte werden so berechnet, dass die Ausführungspfade durch eine Methode gezählt werden. Dabei erhält jede Methode eine initiale Komplexität von eins, zusätzlich erhöht sich die Komplexität um eins für jede Aufteilung des Ausführungspfads durch `if`, `else`, `for` oder `while`. Außerdem wird der Wert für jeden logischen Operator wie `&&` oder `||` in einem logischen Statement um eins erhöht. Für jede Möglichkeit einer `switch`-Anweisung wird die Komplexitätszahl auch um eins erhöht, hinzu kommt je eine Erhöhung um eins bei den Schlüsselwörtern `break`, `goto`, `return`, `throw`, `continue`, `catch` und `except`. Das Ergebnis der Berechnung zeigt Tabelle 10.

	maximale Komplexität	mittlere Komplexität
Straus API Tool	108	5,06
Straus Multitool	19	1,74

Tabelle 10: Code-Komplexität der Programme nach Steve McConnell

Es ist deutlich zu sehen, dass nach dieser Berechnung der Komplexität das Straus Multitool besser abschneidet. Teilweise lässt sich dies darauf zurückführen, dass bei der Durchstich-Implementierung noch nicht alle Eventualitäten berücksichtigt wurden. Trotzdem ist dies durch die große Differenz als gutes Ergebnis zu werten.

Auch die Einrücktiefe lässt Rückschlüsse auf die Komplexität des Quellcodes zu. Die maximale Einrücktiefe des Straus API Tools ist höher als neun (höhere Werte werden nicht gemessen), beim Straus Multitool liegt dieser Wert bei fünf. Die mittlere Einrücktiefe liegt bei 1,83 Einrückungen, während sie beim Straus API Tool

bei 3,99 Einrückungen liegt. Auch hier gilt, dass sich die Komplexität des Straus Multitools mit der vollständigen Implementierung aller Funktionen noch erhöhen wird, insgesamt bestätigen die Ergebnisse zur Verständlichkeit aber die Entscheidung für eine Neuentwicklung.

5.1.4 Zuverlässigkeit (C)

Der Zuverlässigkeit im Sinne der Kriterien Reife, Fehlertoleranz und Wiederherstellbarkeit wird seitens Overdick nur geringe Priorität beigemessen. Trotzdem scheint eine Verbesserung der Zuverlässigkeit des Straus Multitools langfristig sinnvoll. Eine gute Reife des Programms lässt sich auf längere Sicht durch gründliches Testen erreichen. Die Abdeckung durch Tests lässt sich im Gegensatz zur allgemeinen Zuverlässigkeit leicht durch eine statische Analyse messen.

Weitere Aspekte der Zuverlässigkeit lassen sich durch gezieltes Hinzufügen von entsprechenden Funktionen verbessern. Die hierdurch erreichte höhere Zuverlässigkeit wäre allerdings nur durch das Aufzeichnen von Ausfallstatistiken in der Evaluationsphase messbar. Die Verbesserung der Wiederherstellbarkeit scheint zum Beispiel angemessen, wenn man den Aufwand bedenkt, den die Ingenieure mit der Entwicklung der Modelldateien haben. Es ist den Ingenieuren nicht zuzumuten, sich selbst um die doppelte Speicherung der Dateien zu kümmern, die sie mit dem Straus Multitool öffnen wollen. Als Möglichkeit bietet sich eine Dateidopplung beim Öffnen der Datei an; es wäre zu untersuchen, wie sinnvoll sich ein Versionskontrollsystem zur Speicherung von Zwischenständen zur Programmlaufzeit verwenden lässt.

5.1.5 Benutzbarkeit (B)

Die Benutzbarkeit messbar zu beurteilen, fällt schwer. So bemerkt schon Dirk Hoffmann in seinem Buch „Software-Qualität“, dass „die Ergonomieeigenschaften einer Applikation zwar subjektiv zu spüren, aber nur schwer zu quantifizieren“ sind (Hoffmann, 2008 S. 247). Deshalb soll die Benutzbarkeit im Rahmen einer Evaluation beurteilt werden. Bisher steht diese noch aus, weil das Straus Multitool noch nicht ausreichend Funktionen zur Verfügung stellt, um ein aussagekräftiges Ergebnis erreichen zu können.

Es wurden allerdings verschiedene Maßnahmen ergriffen, die Benutzbarkeit zu gewährleisten, die als positiv einzustufen sind: Die Oberfläche zeigt in jedem Kontext die verwendeten Größeneinheiten an, es wurde ein Konzept für den Umgang mit Fehlern etabliert, die Funktionen eines gemeinsamen Kontexts werden in der

Oberfläche gruppiert und es ist ein Benutzerhandbuch geplant. Eine wichtige Hilfe bei der Gestaltung der Oberfläche waren regelmäßige Reviews mit den späteren Benutzern sowie die positiv ausgefallene Bewertung eines externen Usability-Experten, dem das fertige Konzept präsentiert wurde.

5.1.6 Effizienz (B)

Es wurden von Overdick keine quantitativen Vorgaben zur Effizienz gemacht, allein eine hohe Reaktivität („gefühlte Performance“) wurde vorausgesetzt. Im Vergleich zum Straus API Tool konnte für das Straus Multitool eine markante Steigerung der Reaktivität erreicht werden, indem pro Aufgabe nur die Daten aus der Modelldatei gelesen werden, die in diesem Kontext benötigt werden. Es wird kein alles umfassendes Strukturmodell in Objekten abgebildet. Dadurch verteilt sich die Zeit des Einlesens auf unterschiedliche Momente. Für den Benutzer bedeutet dies, dass die Wartezeit für das Öffnen eines Modells sich von 4-8 Sekunden (je nach Modellgröße) auf unter eine Sekunde verringert hat (abhängig von Rechenleistung und Modellgröße). Damit gibt es im Straus Multitool keine spürbare Unterbrechung des Arbeitsflusses. Insgesamt können die Anforderungen an die Effizienz des Straus Multitools also als erfüllt angesehen werden.

5.2 Zukunft des Projekts

Im weiteren Entwicklungsverlauf muss die geforderte Funktionalität noch vollständig implementiert werden. Dazu muss zunächst das Anforderungsdokument in diesen Bereichen genauer spezifiziert werden, da die momentane Version nur Ideen von den Funktionen skizziert.

Einige Aufgaben sind darüber hinaus zu erfüllen:

- Die Benutzeroberfläche sollte durch das Ingenieursteam abschließend evaluiert werden.
- Es müssen ausführliche Testfälle erdacht und implementiert werden, die das Programm und vor allem die sicherheitsrelevanten Abschnitte gründlich testen.
- Ein Benutzerhandbuch muss erstellt werden.

5.3 Abschließende Bewertung

Mit dem Straus Multitool hat die Firma Overdick eine gründlich dokumentierte und solide Architektur erhalten, sowie eine Durchstich-Implementierung, die im

Wesentlichen nur noch funktional ergänzt werden muss. Nach einer gründlichen Test- und Evaluierungsphase kann das Programm in den alltäglichen Arbeitsablauf der Ingenieure übernommen werden und ihnen die Arbeit erleichtern.

LITERATUR

Apache Software Foundation Apache log4net [Online]. - 26. Januar 2012. - http://logging.apache.org/log4net/.

Balzert Helmut Lehrbuch der Software-Technik: Software-Management, Software-Qualitätssicherung, Unternehmensmodellierung [Buch]. - Berlin : Spektrum Akademischer Verlag, 1998.

Bass Len, Clements Paul und Kazman Rick Software architecture in practice [Buch]. - Boston : Addison-Wesley Professional, 2003.

Beck Kent Implementation Patterns [Buch]. - München : Addison-Wesley, 2008.

Booch Grady Object Solutions [Buch]. - Amsterdam : Addison Wesley, 1996.

Campwood Software LCC SourceMonitor [Software] / Hrsg. www.campwoodsw.com. - Burlington : [s.n.]. - 3.2.3.218.

CodePlex Open Source Community Dynamic Data Display [Online]. - 15. November 2011. - http://dynamicdatadisplay.codeplex.com/.

Collins-Sussman Ben, Fitzpatrick Brian W. und Pilato C. Michael Versionskontrolle mit Subversion [Buch]. - Köln : O'Reilly, 2009 .

Feathers Michael C. Effektives Arbeiten mit Legacy Code [Buch]. - Heidelberg : mitp, 2011.

Fowler Martin Refactoring [Buch]. - München : Addison-Wesley, 2005.

Freeman Eric und Freeman Elisabeth Entwurfsmuster von Kopf bis Fuß [Buch]. - Sebastopol : O'Reilly, 2005.

Gamma Erich [et al.] Entwurfsmuster [Buch]. - München : Addison-Wesley, 2004.

Hibernating Rhinos LTD Rhino Mocks [Software]. - Sede Izhak : http://hibernatingrhinos.com/open-source/rhino-mocks.

Hoffmann Dirk W. Software-Qualität [Buch]. - Berlin : Springer, 2008.

Huber Thomas Claudius Windows Presentation Foundation [Buch]. - Bonn : Galileo Press, 2010.

IEEE Computer Society IEEE Std 830-1998 // IEEE Recommended Practice for Software Requirements Specification. - New York : The Institute of Electrical and Electronics Engineers, Inc., 1998.

Krypczyk Veikko Die Arbeit mit WPF-Commands [Journal] // dot.NET Magazin. - Frankfurt am Main : Software & Support Media, 2011. - 6.

Krypczyk Veikko Konzeption und Umsetzung einer Undo-/Redo-Funktion [Journal] // dot.NET Magazin. - Frankfurt am Main : Software & Support Media, 2011. - 11.

Martin Robert C., Martin Micah D. und Wilson-Welsh Patrick FitNesse User Guide [Online]. - 26. Januar 2012. - http://fitnesse.org/FitNesse.UserGuide.

Martin Robert Object Mentor [Online] = The Dependency Inversion Principle. - 26. Januar 2012. - http://www.objectmentor.com/resources/articles/dip.pdf.

McConnell Steve Code Complete [Buch]. - Redmond : Microsoft Press, 1993.

McConnell Steve Code Complete 2 [Buch]. - Redmond : Microsoft Press, 2004.

Microsoft Patterns & Practices Team Microsoft Application Architecture Guide [Buch]. - [s.l.] : Microsoft Press, 2009.

Microsoft Research Moles - Isolation framework for .NET [Software]. - http://research.microsoft.com/en-us/projects/moles/ : [s.n.].

MSDN Library Sprachübergreifende Interoperabilität [Online]. - Microsoft, 2012. - 10. Februar 2012. - http://msdn.microsoft.com/de-de/library/a2c7tshk.aspx.

Osherove Roy The Art of Unit Testing [Buch]. - Heidelberg : mitp, 2010.

Poole Charlie, Cansdale Jamie und Feldman Gary NUnit.org [Online]. - 26. Januar 2012. - http://www.nunit.org/.

Siedersleben Johannes Moderne Softwarearchitektur [Buch]. - Heidelberg : dpunkt, 2004.

Spillner Andreas und Linz Tilo Basiswissen Softwaretest [Buch]. - Heidelberg : dpunkt, 2003.

Starke Gernot Effektive Software-Architekturen [Buch]. - München : Hanser, 2011.

Starke Gernot und Hruschka Peter Software-Architektur kompakt [Buch]. - Heidelberg : Spektrum Akademischer Verlag, 2011.

Strand7 Pty Ltd Strand7 website [Online]. - 2. Oktober 2011. - http://www.strand7.com/.

Straus7 Software API Manual. - Sydney : Strand7 Pty Limited, 2010.

Szyperski Clemens, Grundtz Dominik und Murer Stephan Component software [Buch]. - Harlow : Pearson Education, 2002.

TIOBE Software TIOBE Programming Community Index [Online]. - 12. Februar 2012. - http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html.

Wikipedia Application checkpointing [Online]. - 16. Januar 2012. - http://en.wikipedia.org/w/index.php?title=Application_checkpointing&oldid=468352446.

Wirfs-Brock Rebecca und McKean Alan Object Design [Buch]. - Boston : Pearson Education, 2002.

A ANHANG

Der Anhang zu dieser Arbeit ist in elektronischer Form auf einer CD abgelegt. Diese kann beim Prüfer Prof. Dr. Thomas Lehmann eingesehen werden.

A.1. LISTE DER ANFORDERUNGEN

In diesem Kapitel werden die bei der Anforderungsanalyse ermittelten Anforderungen aufgeführt. Die Notation der Anforderungen ist in Kapitel 1.5 erklärt. Für einen Überblick über die geplanten Funktionen des Straus Multitools sollte auch zuerst dort nachgelesen werden. Domänenspezifische Begriffe können in Anhang A.2 „Glossar der Fachdomäne" nachgeschlagen werden.

Die Struktur dieses Kapitels orientiert sich an den Vorschlägen der Norm IEEE 830 (IEEE Computer Society, 1998). Deshalb werden zuerst Angaben über die externen Schnittstellen des Straus Multitools gemacht, anschließend werden die funktionalen und die nicht-funktionalen Anforderungen beschrieben. Der Abstraktionsgrad der Anforderungen für eine spätere Umsetzung wurde auf Lastenheftniveau gehalten. Anforderungen für die Durchstich-Implementierung wurden auf Pflichtenheftniveau erhoben. Alle Anforderungen werden auf Grund der zeitlichen Beschränkung der Arbeit nur so detailliert beschrieben, wie für den Architekturentwurf und die Durchstich-Implementierung notwendig ist.

A.1.1. EXTERNE SCHNITTSTELLEN

Das Straus Multitool soll mit externen Programmen zusammenarbeiten. Dies wird in diesem Abschnitt beschrieben. Die Systemgrenzen der Anwendung werden in Kapitel 2, Abbildung 1 auf Seite 8 beschrieben.

Bisher ist die Zusammenarbeit mit der Programmierschnittstelle von Straus7 und mit einer Bibliothek zur Generierung von Wellenlasten fest eingeplant. In Zukunft könnte sich dieses Kapitel um Anforderungen zu Schnittstellen mit anderen Entwicklungsprogrammen für Strukturmodelle oder Programme für technische Zeichnungen erweitern.

REQ-023	Straus API	Durchstich Straus API Tool

Das Straus Multitool soll die Funktionen der Straus-API nutzen. Die API stellt Funktionen zur Verfügung, die den Umgang mit Straus-Modelldateien ermöglichen. Dazu gehört das Öffnen, Schließen und Speichern der Dateien, aber auch

das Interpretieren des Dateiinhalts. Ein Handbuch zu den Funktionen der Straus-API findet sich im Installationsverzeichnis von Straus7 im Unterordner „API".

REQ-024	Waveloads.dll	später Straus API Tool

Das Straus Multitool soll mit der Bibliothek Waveloads.dll zusammenarbeiten können. Die Bibliothek wird von der Forschergruppe GIGAWIND der Universität Hannover zur Verfügung gestellt. Sie stellt Daten zur Generierung von Wellenlasten bereit. Eine Dokumentation dazu findet sich im Internet[2].

A.1.2. Funktionale Anforderungen

In diesem Abschnitt sollen die funktionalen Anforderungen des Programms beschrieben werden. Dabei werden die Programmfunktonen in Funktionengruppen eingeteilt, denen die Anforderungen zugeordnet werden.

A.1.2.1. Grafische Benutzeroberfläche

In diesem Abschnitt werden Anforderungen aufgeführt, die der Benutzeroberfläche zugeordnet sind. Anforderungen an die Oberfläche, die speziellen Modulen zugeordnet sind, werden in den entsprechenden Abschnitten zu diesen Modulen beschrieben.

REQ-014	Modellfenster	Durchstich KO

Das Straus-Modell soll als grafische Darstellung in einem eigenen Fenster angezeigt werden können. Dieses Fenster sollte unabhängig vom Hauptfenster des Straus Multitools sein, damit man es z. B. auf einen zweiten Monitor verschieben kann und mehrere Ansichten parallel sehen kann.

Die Straus-API bietet die Möglichkeit, ein Modellfenster zu erzeugen, das sich in der Ansicht nicht von der Darstellung innerhalb von Straus7 unterscheidet. Dieses Modellfenster bringt auch einige Funktionen mit, die die Darstellung den Bedürfnissen des Benutzers anpassen. Das Modell kann z. B. gedreht werden und verschiedene Lastfälle können als Vektoren eingeblendet werden. Dieses Modellfenster ist mit dieser Anforderung gemeint.

REQ-056	Sprache: Englisch	Durchstich FLO

[2] http://www.gigawind.de/fileadmin/gigawind/downloads/waveloads/UsermanualWL.pdf

Die Sprache der Hilfe und der Oberfläche des Programms soll Englisch sein. Internationale Mitarbeiter sollen das Programm ebenfalls verstehen und bedienen können (in internationalen Zweigstellen des Unternehmens wird Englisch gesprochen). Eine Mehrsprachigkeit des Programms wird zunächst nicht angestrebt.

REQ-038	Verschiedene Pfade pro Funktion	⌛ später
		RE, FLO

Zu den Programm-Funktionen, wie z. B. Öffnen einer Modelldatei, sollen mehrere mögliche Pfade führen. Dazu gehören Symbolleisten-Elemente, Einträge in Menü und ggf. Kontextmenü sowie die Ansteuerung über Tastenkürzel.

REQ-053	Tabellenfunktionen	⌛ später
		RE

An mehreren Stellen im Straus Multitool müssen Daten mit Hilfe von Tabellen dargestellt werden (z. B. bei *REQ-019* und bei einigen Modulen). Die angezeigten Tabellen sollen ein umfangreiches Kontextmenü aufweisen, mit dessen Hilfe Bereiche der Tabelle kopiert, gesucht, gefiltert, verändert und konfiguriert werden können. Folgende Funktionen sind geplant:

- **Copy with Header Text**: Die angeklickte Spalte bzw. die markierten Spalten werden so in die Zwischenablage kopiert, dass sie mit der Spaltenüberschrift in ein Excel-Sheet eingefügt werden können.
- **Copy without Header Text**: Funktion wie bei „Copy with Header Text" nur ohne die Spaltenüberschrift.
- **Paste**: Ein einzelner Wert aus als Zeichenkette oder mehrere Zellen, die aus einem Excel-Sheet in die Zwischenablage kopiert wurden, werden an der angeklickten Stelle in die Tabelle eingefügt.
- **Select All**: Markiert alle Zellen der Tabelle.
- **Select Row**: Markiert alle Zellen der angeklickten Zeile.
- **Select Column**: Markiert alle Zellen der angeklickten Spalte.
- **Find**: Sucht die erste Fundstelle eines einzugebenden Werts in der Tabelle.
- **Find Next**: Sucht den zuletzt gesuchten Wert unterhalb der letzten Fundstelle.
- **Find & Replace**: Sucht einen einzugebenden Wert in der Tabelle und ersetzt ihn durch einen weiteren einzugebenden Wert.
- **Visualize Values**: siehe *REQ-054*
- **Filtering…**: siehe *REQ-055*
- **Show Graph**: siehe *REQ-043*

REQ-043	Daten als Funktionen anzeigen	später Straus API Tool

In Tabellenansichten sollen Spalten ausgewählt werden können und die enthaltenen Daten sollen als Funktion voneinander grafisch dargestellt werden können. Das Auslösen dieser Funktion erfolgt zum Beispiel über einen Eintrag im Kontextmenü (siehe *REQ-053*).

REQ-054	Werte in Tabelle visualisieren	später RE

Mit dem Befehl „Visualize Values" des Kontextmenüs für Tabellen (*REQ-053*) sollen alle Werte in den markierten Spalten mit farbigen Balken hinterlegt werden, deren Länge und Farbe den Wert der Zelle im Vergleich zu allen anderen Zellen dieser Spalte abbilden. Dabei soll ein kurzer dunkelblauer Balken das Minimum einer Spalte markieren und ein langer roter Balken das Maximum. Die Werte, die zwischen diesen liegen, werden farblich und in der Länge interpoliert.

REQ-055	Tabellenwerte filtern	später RE

Mit dem Befehl „Filtering…" (vgl. Kontextmenü in *REQ-053*) soll eine Möglichkeit gegeben werden, die Zeilen der Tabelle nach bestimmten Kriterien zu filtern. Momentan ist die Idee, dies über eine Konsoleneingabe zu realisieren.

A.1.2.2. Modellverwaltung

In diesem Abschnitt werden Anforderungen beschrieben, die sich mit der Verwaltung des Straus-Modells beschäftigen. Dies beinhaltet lesende und schreibende Zugriffe auf Modelldateien. Die Straus-API stellt hilfreiche Funktionen für diese Anforderungen zur Verfügung (vgl. *REQ-023*).

REQ-019	Geometriedaten einlesen	Durchstich Straus API Tool

Das Programm soll die Geometriedaten eines Straus-Modells (Dateiendung `.st7`) über die Straus-API einlesen, interpretieren und in einer geeigneten Form anzeigen können. Die Anzeige kann z. B. eine Tabellenansicht von Daten oder die 3D-Ansicht des Modells sein (vgl. *REQ-014*) – je nach Anwendungsbereich.

REQ-026	Modell verändern und speichern	Durchstich Straus API Tool

Das Programm soll das Modell anpassen können und in der Lage sein, es verändert abzuspeichern.

REQ-050	Modell schließen	Durchstich FLO

Eine Modelldatei soll geschlossen werden können, damit ein anderes Modell geöffnet und bearbeitet werden kann, ohne dass das Programm neu gestartet werden muss.

A.1.2.3. LASTENAUFBRINGUNG

In diesem Abschnitt soll auf die Anforderungen eingegangen werden, die sich mit der Aufbringung von Lasten auf ein Straus-Modell beschäftigen. Die Lasten werden ebenso wie die Geometriedaten in der Straus-Modelldatei gespeichert. Zugriff darauf bietet die Straus-API, siehe *REQ-023*.

REQ-020	Lastfälle verwalten	Durchstich Straus API Tool

Die Lastfälle und Lastfallkombinationen zu einem Modell sollen geladen, gespeichert und angezeigt werden können. Außerdem sollen neue Lasten zu einem Lastfall hinzugefügt werden können und neue Lastfälle sollen angelegt werden können.

REQ-046	Elemente für Lasten auswählen	Durchstich Straus API Tool

Für die Aufbringung von Lasten sollen Elemente ausgewählt werden, auf denen die Lasten simuliert werden sollen. Diese Elemente können nach den Kriterien „Group“, „Beam Properties“, „Plate Properties“ und „ID“ ausgewählt werden.

Zunächst sollen die Elemente über eine Auflistung der Elemente ausgewählt werden können. Die Gruppen können dabei drei Zustände aufweisen: selektiert/nicht selektiert/teilweise selektiert – je nachdem, ob alle Elemente, keine Elemente oder nur einige Elemente aus der Gruppe selektiert wurden. Die einzelnen Elemente können nur selektiert oder nicht selektiert sein.

Langfristig sollen die Elemente auch grafisch durch Markieren in der Modellansicht ausgewählt werden können.

Die Auswahl soll jeweils der Lastengenerierung durch das Windloads-Modul und das Waveloads-Modul vorgelagert sein. Die Module sollen dann nur mit den selektierten Elementen arbeiten.

REQ-040	Lastfall-Kombinationen generieren	⌛ später RE

Aus angelegten Lastfällen sollen automatisch Lastfall-Kombinationen generiert werden können. Dazu kann für jeden Lastfall eine Gruppenzugehörigkeit (Lastgruppe) festgelegt werden und das Attribut S/V (ständig / veränderlich) gesetzt werden können. Eine beispielhafte Tabelle ist in Abbildung 25 dargestellt. Aus den Angaben in der Tabelle sollten dann automatisch die Lastfall-Kombinationen

	S/V	Lastgruppe
Lastfall 1: Eigengewicht	S	
Lastfall 2: Wind x	V	1
Lastfall 3: Wind y	V	1
Lastfall 4: Welle x	V	2
Lastfall 5: Welle y	V	2

Abbildung 25: Eingabe zum Generieren von Lastfall-Kombinationen (Beispiel)

- LF1 + LF2 + LF4
- LF1 + LF2 + LF5
- LF1 + LF3 + LF4
- LF1 + LF3 + LF5

generiert werden. Damit wird das Eigengewicht immer berücksichtigt und es werden alle möglichen Kombinationen generiert, mit Rücksicht darauf, dass Lastfälle innerhalb einer Lastgruppe nicht gemeinsam auftreten können (entweder kommt der Wind aus der x-Richtung oder aus der y-Richtung).

A.1.2.3.1. Waveloads

Eine wichtige Funktion des Straus Multitools soll das Generieren von Wellenlasten sein, die auf das Modell aufgebracht werden können. Die damit zusammenhängenden Anforderungen werden in diesem Abschnitt beschrieben.

REQ-045	Waveloads-Lastfälle generieren	⌛ später Straus API Tool

Das Waveloads-Modul soll Lastfälle generieren können. Dabei sollen verschiedene Parameter frei einstellbar sein. Die Parameter sind dem Straus API Tool zu entnehmen und im Detail mit den Mitarbeitern von Overdick abzusprechen.

REQ-007	Konfigurationen speichern	⌛ später Meetingprotokoll vom 24. 5. 2011

Eine einmal eingestellte Kombination von Parametern für die Berechnung der Windlasten sollte als Konfiguration speicherbar sein. Eine so gespeicherte Konfiguration sollte einfach auch wieder geladen und durch einen eindeutigen Namen identifiziert werden können.

REQ-037	Extreme Lasten auswählen	später Straus API Tool, NW

Nach Generierung der Lastfälle und Berechnung der Ergebnisse sollen die extremsten Lasten im Modell als neue Kombination abgespeichert werden.

A.1.2.3.2. WINDLOADS

Ein weiteres Modul soll Windlasten generieren können. Die zugehörigen Anforderungen werden in diesem Abschnitt beschrieben.



REQ-022	Windloads-Lastfälle generieren (DNV)	Durchstich Straus API Tool

Das Windloads-Modul soll Lastfälle nach der Norm DNV-RP-C205 generieren können. Dabei sollen verschiedene Parameter frei einstellbar sein (in Klammern stehen die voreingestellten Werte):

- Reference Height (10m)
- Average Time (3 sec)
- Reference Time (10 sec)
- Reference Wind Speed (30 m/sec)
- **Wind Spreading (0°)**
- **Density of Air (1,2260 kg/m³)**
- **Gravity Direction (Z-Direction, negative axis)**
- **Translation (0 mm)**
- **Marine Growth Thickness (10 cm)**
- **Height of natural cover – minimal height an maximal height**

Die Parameterkombinationen sollen nicht speicherbar sein.

REQ-022a	Windloads-Lastfälle generieren (DIN)	später Straus API Tool

Das Windloads-Modul soll Lastfälle nach DIN-1055 generieren können. Dabei sollen die folgenden Parameter frei einstellbar sein (in Klammern stehen die voreingestellten Werte):

- Geländekategorie (0)
- Windzone (3)
- v_{ref} (30 m/sec)
- q_{ref} (0,56 kN/m²)
- Probability of Exceedence (0,98)

Hinzu kommen die fett gedruckten Parameter, die auch in *REQ-022* aufgeführt werden. Die Kombination der Parameter soll nicht speicherbar sein.

REQ-022b	Windlasten anzeigen	Durchstich Straus API Tool

Die generierten Windlasten können in einer Tabellenansicht überprüft werden. Eine Grafik zeigt die aus den eingestellten Parametern resultierenden Windlasten.

REQ-022c	Windlasten speichern	Durchstich Straus API Tool

Der Benutzer kann die berechneten Lasten in einem neuen oder einem vorhandenen Lastfall in der Modelldatei abspeichern (siehe *REQ-020*).

A.1.2.3.3. Weitere Lastprüfungen

Ähnlich wie die Module Waveloads und Windloads sind Module wie der Beam-Buckling-Check und der Joint-Check auch Prüfungen der Modellstabilität durch Lastaufbringung. Die Implementierung dieser beiden Module ist für später geplant.

REQ-028	Beam-Buckling-Check	später Straus API Tool

Das Programm soll die Berechnung eines Beam-Buckling-Checks nach Norm EC3 oder DIN 18800 durchführen können und ein korrektes Ergebnis dieser Rechnung zurückliefern.

REQ-029	Joint-Check	später Straus API Tool

Das Programm soll die Berechnung eines Joint-Checks durchführen können und ein korrektes Ergebnis dieser Rechnung zurückliefern.

A.1.2.4. Ergebnisse

Langfristig soll das Straus Multitool möglicherweise auch mit Ergebnisdateien umgehen können. Das sind Dateien, die vom Programm Straus erstellt werden und die das Ergebnis einer bestimmten Berechnung an einem bestimmten Modell enthalten. Da Straus zum Umgang mit Ergebnissen aber auch viele gute eigene Funktionen anbietet, ist noch nicht klar, ob die Anforderungen in diesem Kapitel umgesetzt werden.

REQ-021	Ergebnisdaten verwalten	⌛ später Straus API Tool

Straus-Ergebnisdateien sollen gelesen und geschrieben werden können. In den Ergebnisdateien befindet sich das Ergebnis jeweils einer Berechnung in Straus.

REQ-031	Modell als vorverformte Struktur	⌛ später Straus API Tool

Das Modell soll als vorverformte Struktur speicherbar sein. Die ermittelten Ergebnisse sollen also als neues (verformtes) Modell gespeichert werden können.

REQ-005	Automatischer Bericht	⌛ später Meetingprotokoll vom 24. 5. 2011

Aus den Ergebnissen soll ein automatischer Bericht generiert werden können. Dieser soll grafische Darstellungen sowie Textelemente, Größen mit Einheiten und Tabellen enthalten können.

A.1.2.5. Kleinere Hilfsfunktionen

Das Straus-Multitool soll eine Reihe kleinerer Hilfsfunktionen anbieten. Deren Funktionen werden in diesem Abschnitt kurz angerissen.

REQ-030	Auffinden von unverbundenen Stäben	⌛ später JHU

Die Software soll bei importierten Modellen (siehe auch *REQ-036*) mit Hilfe eines Zielwinkels unverbundene Stäbe und Platten aufspüren. Dabei soll die Suche auf einzelne Member-Gruppen im Modell einschränkbar sein. Member-Gruppen sind als Elemente im Modell gespeichert und verweisen auf die einzelnen Elemente (Stäbe, Knoten, …) in ihnen.

REQ-033	Selektieren von Stäben bestimmter Länge	⌛ später Straus API Tool

Die Software kann aus dem Modell alle Stäbe oberhalb einer bestimmten Länge selektieren. Diese können auf Wunsch unterteilt werden. Wenn die Stäbe vorher zu lang waren, wird mit diesem Schritt die Berechnung des Modells erst möglich.

REQ-035	Clean Mesh	⌛ später Straus API Tool

Räumlich übereinander liegende Knoten, die eigentlich eine Verbindung sein sollten, können zu einem Knoten verschmolzen werden.

REQ-036	Tekla-Import	später SNA

Das Programm ist in der Lage, Dateien des Programms Tekla zu importieren und in ein Straus-Modell zu konvertieren.

REQ-018	Makros aufnehmen	später RE

Die einzelnen Schritte, die im Verlauf einer Session durchgeführt werden, sollten auf Wunsch des Nutzers hin aufgezeichnet werden können (wie ein Makro), um später automatisch wiederholt werden zu können.

A.1.2.6. Administrative Funktionen

In diesem Abschnitt sollen Anforderungen zur Programmverwaltung beschrieben werden.

REQ-006	Logging	Durchstich RE, Straus API-Tool

Das Programm soll eine Log-Datei generieren, in der man den Status von Lade- oder Berechnungsvorgängen sehen kann und auch, wo möglicherweise Fehler aufgetreten sind.

Es soll klar sein, an welcher Stelle ein Vorgang möglicherweise abgebrochen wurde. Dazu sollen die Zeilen mit Datum und Uhrzeit versehen werden.

Die Module sollten der Übersichtlichkeit wegen in die gleiche Log-Datei schreiben wie das Hauptprogramm. Die Einträge sollten aber entsprechend gekennzeichnet werden, sodass klar ist, in welcher Komponente ein möglicher Fehler aufgetreten ist:


```
2011-11-21 09:32:24    Modulname: Vorgang und Status
```

Abbildung 26: Beispielzeile für die Log-Datei

REQ-016	Automatisches Update	später RE, FLO

Es soll eine Funktion geben, die beim Starten des Programms abgleicht, ob die aktuelle Version des Programms vorliegt. Ist dies nicht der Fall, soll die neue Version des Programms installiert werden.

REQ-013	Lizenz als Programmschutz	später RE

Das Programm soll nur mit einem Passwort oder einer Lizenz benutzbar sein um den Nutzerkreis auf Mitarbeiter von Overdick einzuschränken.

REQ-057	Undo/Redo	Durchstich FLO, KO

Änderungen an der Modelldatei oder angezeigten (ungespeicherten) Daten in Tabellen sollen rückgängig gemacht und ebenso wieder hergestellt werden können.

A.1.3. Anforderungen an Performance

In diesem Kapitel werden Aussagen zur gewünschten Performance des Straus Multitools gemacht. Da es in diesem Programm kein zeitkritisches Verhalten gibt, geben die Aussagen nur eine Richtung vor. Ob die Programmperformance und die Reaktivität den Wünschen der Benutzer entsprechen, muss in einer Evaluation ermittelt werden.

REQ-015	Ladezeiten	später RE

Das Laden von Datenmengen von der Festplatte oder aus dem Arbeitsspeicher soll so programmiert werden, dass auch bei großen Mengen die Oberfläche benutzbar bleibt.

Dabei kann zwischen der Handhabung großer Datenmengen unterschieden werden und der Überlegung der Notwendigkeit einer vollständigen Darstellung für den Anwender. In vielen Fällen mag vielleicht eine Zusammenfassung der Datenmenge mehr helfen. Für die Handhabung großer Datenmengen ist *REQ-015a* eine Möglichkeit.

REQ-015a	Dynamisches Nachladen	später RE

Beim Laden von Listen, z.B. in eine Tabelle, sollen die Daten in Abschnitte unterteilt werden, die nacheinander geladen werden. Dies ist zum Beispiel beim `DataGridView` mit dem „Virtual Mode“ möglich.

A.1.4. Design Constraints

In diesem Kapitel werden Anforderungen beschrieben, die an das Design des Programms gestellt werden. Es handelt sich hierbei um von Overdick-Mitarbeitern gewünschte Eigenschaften.

REQ-001	Modulare Bauform	Durchstich RE, FLO

Verschiedene Berechnungen sollen als Module über Assemblies eingebunden werden. Durch eine derartige Struktur nehmen die Erweiterungen über eine einheitliche Schnittstelle Kontakt zum Hauptprogramm auf, ändern den Quellcode aber nicht.

REQ-002	Trennung von GUI und Funktionen	Durchstich FLO

Die Anwendung trennt die Zuständigkeiten der grafischen Oberfläche von der Funktion über Einhaltung des Model-View-Controller-Patterns oder einer vergleichbaren Aufteilung.

A.1.5. Dokumentation

Die Anforderungen an die Dokumentation der Programmfunktionen und der Implementierung werden in diesem Kapitel beschrieben.

REQ-003	Benutzerhandbuch	Durchstich Meetingprotokoll vom 24. 5. 2011

Es soll ein möglichst vollständiges und hilfreiches Benutzerhandbuch entstehen. Darin sollen mindestens die Installation (wenn sie nicht vollständig geführt ist) und die gängigsten Anwendungsschritte erklärt werden.

Ein nicht an der Entwicklung beteiligter Anwender soll das Programm nur mit Hilfe des Handbuchs installieren und ausprobieren können.

REQ-008	Quellcode-Dokumentation	Durchstich KO

Aus der Quellcode-Dokumentation soll hervorgehen, wie das Programm funktioniert, wozu die einzelnen Abschnitte gedacht sind und wie die Architektur aufgebaut ist (ggf. mit einem Zusatz-Dokument).

Zukünftige Entwickler sollen sich einfach im Code zurechtfinden: Ihn verstehen und Änderungen einpflegen können.

REQ-009	Dokumentation der Module	Durchstich NW

Für jede Funktion soll eine Beschreibung existieren, aus der hervorgeht, welches Verfahren verwendet wird, welche Einschränkungen dieses Verfahren mit sich bringt und welche Randbedingungen für die Rechnung angenommen werden.

Im Programmhandbuch und ggf. an weiteren Stellen im Programm werden entsprechende Angaben gemacht.

A.1.6. Benutzbarkeit

In diesem Kapitel werden von Overdick-Mitarbeitern formulierte Anforderungen an die Benutzbarkeit des Straus Multitools dokumentiert. Teilweise konnten diese nicht genau spezifiziert werden, werden aber als Leitgedanke aufgenommen und berücksichtigt.

REQ-012	Gute Benutzbarkeit	Durchstich KO

Das Programm soll für die Nutzer verständlich und eindeutig sein. Ein neuer Benutzer soll sich relativ schnell einfinden können.

REQ-017	Einfache Installation	später TW, FLO

Die Installation des Programms soll einfach und intuitiv sein. Das selbstständige Kopieren von DLL-Dateien und Eintragen von Pfaden soll nicht nötig sein, stattdessen soll sich die Installation am Windows-Standard für Installationsroutinen orientieren.

Anmerkung: Diese Anforderung war zunächst für den Durchstich vorgesehen. Dies hat sich aufgrund des geringen Funktionsumfangs als unnötig herausgestellt.

REQ-004	Physikalische Einheiten in der GUI	Durchstich Meetingprotokoll vom 24. 5. 2011

Es soll zu jeder Zeit deutlich sein, welche Einheiten verwendet werden. Dazu ist jede Ein- oder Ausgabe von Zahlenwerten ist (sofern der Wert nicht eindeutig als einheitenlos erkannt wird) mit einem Hinweis auf die verwendete Einheit zu versehen.

REQ-011	Übersichtliche Ergebnisdarstellung	später / KO

Die Darstellung der Ergebnisse soll übersichtlich und informativ sein. Dazu können Tabellen, Listen, Text oder auch Hüllkurven verwendet werden. Die Übersichtlichkeit muss mit Hilfe der zukünftigen Anwender evaluiert werden.

A.1.7. Qualitätsanforderungen

In diesem Kapitel finden sich die Qualitätsanforderungen, wie sie mit Overdick vereinbart wurden. Weitere Qualitätsaspekte, die sich in der Recherche-Phase ergeben haben, werden in Kapitel 2.1 näher beschrieben.

REQ-039	Berechnungen korrekt	Durchstich / FLO

Zum Zweck des Programms gehört es, den Nachweis zu erbringen, dass ein entwickeltes Modell einschlägigen Normen entspricht. Dazu werden Lasten generiert, mit denen in einer Simulation das Modell belastet wird um die Stabilität zu testen. Aus diesem Grund ist es sehr wichtig, dass die programminternen Rechnungen korrekt sind, da Fehler schlimme Folgen haben könnten. Daher sollen alle Module, die Berechnungen ausführen, besonders gründlich mit Hilfe von automatisierten Unit-Tests auf Korrektheit geprüft werden.

REQ-052	Stabilität	Durchstich / FLO

Das Modell und die Ergebnisse müssen sich zu jedem Zeitpunkt während der Programmlaufzeit in einem konsistenten Zustand befinden. Wenn ein Befehl fehlschlägt oder das Programm unsachgemäß beendet wird, sollte die Datei, an der gearbeitet wurde, nicht irreparabel beschädigt sein.

A.2. Inhalt der CD

Die Inhalte der CD sollen im Folgenden vorgestellt werden. Die CD enthält abgespeicherte Online-Literaturquellen und den Quellcode der Durchstich-Implementierung.

A.1.1. Online-Quellen aus dem Literaturverzeichnis

Die online abgerufenen Quellen wurden aufgrund ihrer Flüchtigkeit in dem Zustand gespeichert, den sie bei Abruf der zitierten Information hatten. Wenn allgemein auf eine Internetseite als Herkunft eines Projekts verwiesen wird, wurde die Seite nicht als Quelle auf die CD aufgenommen. Auch auf das Aufnehmen von verwendeter Open-Source-Software auf die CD wurde verzichtet. Die für das Straus Multitool verwendeten Bibliotheken sind im Quellcode-Verzeichnis abgelegt (siehe Kapitel 0). Die Quellen wurden als PDF gespeichert, obwohl sie dadurch in ihrer Form verändert wurden, da ansonsten keine einfache Konservierung des Zustands möglich war. Die Dateien sind im Order „Online-Quellen“ auf der CD abgelegt.

(Martin)	*The Dependency Inversion Principle*	`dip.pdf`
(MSDN Library, 2012)	*Sprachübergreifende Interoperabilität*	`Interoperabilität.pdf`
(TIOBE Software)	*TIOBE Programming Community Index*	`Tiobe.pdf`
(Wikipedia)	*Application Checkpointing*	`checkpointing.pdf`

A.1.2. Quellcode des Projekts

Der vollständige Quellcode der Durchstich-Implementierung befindet sich auf der CD im Verzeichnis „Quellcode“. Die Ordnerstruktur ist in Abbildung 27 dargestellt und wird im Folgenden erläutert.

Der Quellcode ist pro Komponente in einem eigenen Microsoft Visual Studio Projekt organisiert, damit der Überblick beim Hinzufügen von Testprojekten trotzdem gewahrt bleibt. Mit einer Ausnahme sind die in der Abbildung rechts dargestellten Ordner jeweils einer Komponente zugeordnet. Lediglich der Ordner „StrausMultitool“ enthält sowohl die `Initialization`-Komponente als auch die `Ambience`-Komponente (Projekt „StrausMultitool“). In jedem Ordner befindet sich eine Visual-Studio-Projektmappe, in der die rechts als Papiere dargestellten Projekte organisiert sind.

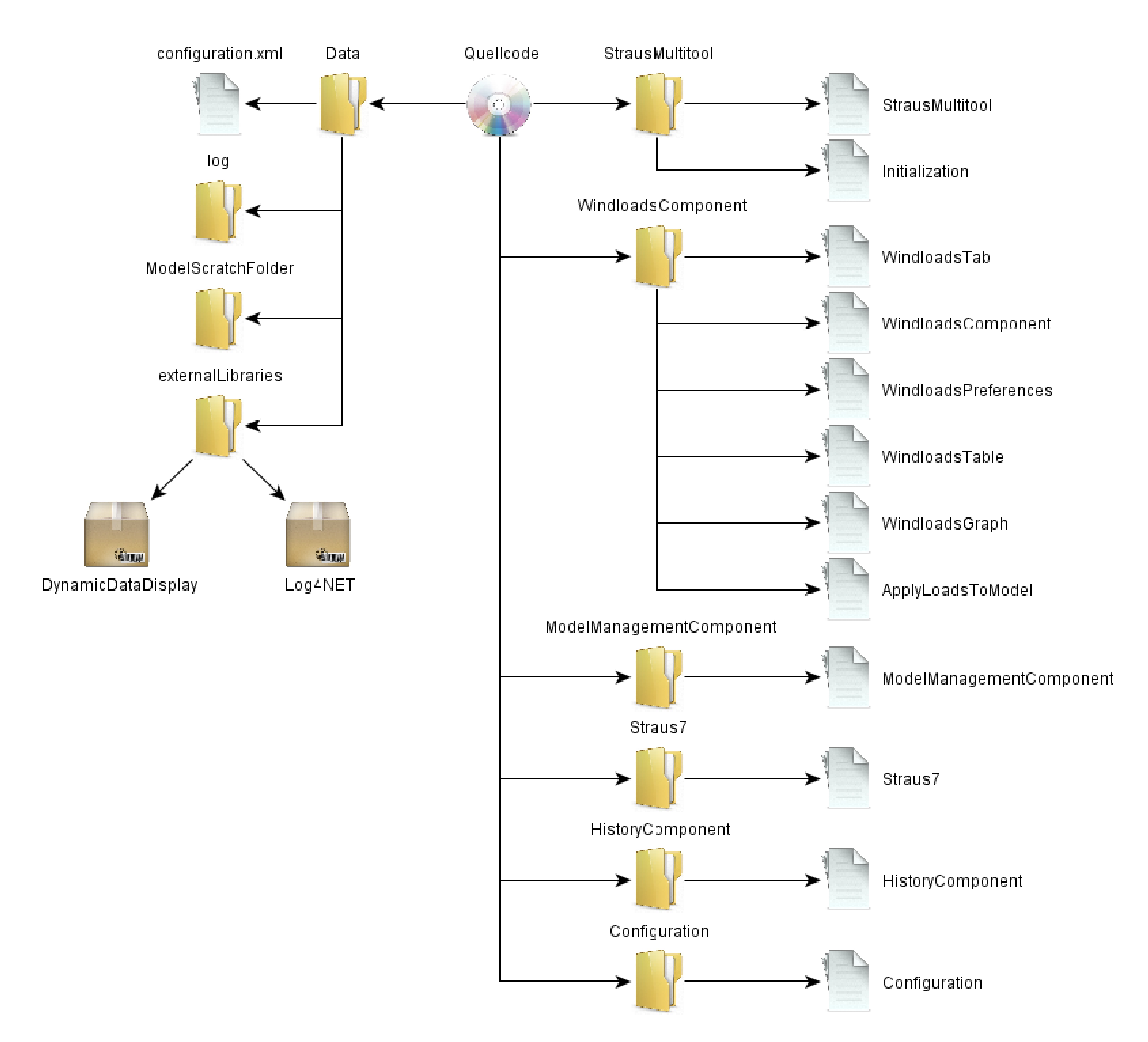


Abbildung 27: Struktur des Quellcodes[3]

Zusätzlich zu den dargestellten Projekten sind Testprojekte angelegt worden. Jedes Projekt wird beim Kompilieren zu einer Bibliothek übersetzt. Diese werden wiederum von anderen Projekten verwendet.

Links in der Darstellung ist der Ordner „Data“ abgebildet. In ihm wurden zusätzliche Daten abgelegt. Dazu zählen externe Bibliotheken, die Protokolldateien, die von Log4NET angelegt werden, das temporäre Verzeichnis für die Straus-API und die Konfigurationsdatei.

[3] Symbole stammen von `http://www.creativefreedom.co.uk/icon-design-info/free-windows-7-icons/` und sind frei verfügbar.

A.2. Glossar der Fachdomäne

Attribut Jedes →*Modellelement* hat eine bestimmte Konfiguration, die die Eigenschaften der Elemente festlegt. Beispielsweise definiert ein Stab-Attribut eine bestimmte Dicke, ein Material und ein Profil des →*Stabs*.
Eine bestimmte Konfiguration wiederholt sich innerhalb eines Modells sehr häufig, weshalb eine bestimmte Konfiguration als Attribut gespeichert wird. Aus diesem Grund ist jedem Element ein Attribut zugeordnet.

Beam →*Stab*

Brick Dreidimensionales →*Modellelement*, das von mindestens acht →*Knoten* definiert und begrenzt wird.

Finite-Elemente-Methode Ein numerisches Verfahren zur Lösung von partiellen Differentialgleichungen. Sie ist ein weit verbreitetes modernes Berechnungsverfahren im Ingenieurwesen und ist das Standardwerkzeug bei der Festkörpersimulation.

Knoten Verbindungselemente zwischen anderen →*Modellelementen* wie →*Stäben* oder →*Platten*.

Last Eine veränderliche oder bewegliche Einwirkung auf ein Bauteil, zum Beispiel infolge Personen, Einrichtungsgegenständen, Lagerstoffen, Maschinen oder Fahrzeugen.

Lastfall Angabe von →*Lasten* und Verformungen, die gleichzeitig einwirken. Der Begriff wird auch als Synonym für die in den Normen geregelten Lasten bzw. Einwirkungen, verwendet, welche auf die Konstruktion bzw. den Aufbau einwirken und diese beeinflussen können.

Lastfall-Kombination Zusammenfassung von mehreren →*Lastfällen* in →*Straus7*. Zur einfacheren Handhabung können mehrere Lastfälle linear zu einem neuen Lastfall kombiniert werden.

Load →*Last*

Load Case →*Lastfall*

Load Case Combination →*Lastfall-Kombination*

Member, Modellelement Oberbegriff für →*Knoten,* →*Stäbe,* →*Platten* und →*Bricks.*

Node →*Knoten*

Plate, Platte Zweidimensionales →*Modellelement*, das von mindestens drei →*Knoten* definiert und begrenzt wird.

Property →*Attribut*

Stab Eindimensionales →*Modellelement*, das von mindestens zwei →*Knoten* definiert und begrenzt wird.

Strand7 Strand7 ist ein Desktopprogramm, das in einer gemeinsamen Umgebung strukturelle Modelle analysiert und Ergebnisse von Lastfällen mit Hilfe der →*Finite-Elemente-Methode* berechnet.

Straus7 Name von →*Strand7* für den europäischen Markt

Versicherung der Selbstständigkeit

Hiermit versichere ich, dass ich die vorliegende Arbeit im Sinne der Prüfungsordnung nach §16(5) APSO-TI-BM ohne fremde Hilfe selbstständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen habe ich unter Angabe der Quellen kenntlich gemacht.

Hamburg, 15. Februar 2012 Friederike Löwe